# HUGGING FACE HANDBOOK

## The AI Researcher's Guide to Using Google Colab

# HUGGING FACE HANDBOOK

*The AI Researcher's Guide to Using Google Colab*

2026 Edition

**Jekardah Tech Review Desk**
By Romi Nur Ismanto
rominur@gmail.com
**jekardah.com**

# Copyright and Disclaimer



DISCLAIMER OF LIABILITY: The information in this book is provided on an “as is” basis for educational purposes only. The author and Jekardah Tech Review Desk shall not be held liable for any damages arising from the use of information contained in this book. Every effort has been made to ensure accuracy, but the rapidly evolving nature of the AI field means that some details, including API syntax, model availability, and pricing, may change after publication. Readers are encouraged to consult official documentation at huggingface.co and colab.research.google.com for the most current information.

TRADEMARK NOTICE: Hugging Face and the Hugging Face logo are trademarks of Hugging Face, Inc. Google, Google Colaboratory, Google Colab, Google Drive, TensorFlow, and related marks are trademarks of Google LLC. PyTorch is a trademark of Meta Platforms, Inc. NVIDIA, CUDA, and Tesla are trademarks of NVIDIA Corporation. OpenAI and Whisper are trademarks of OpenAI, Inc. All other product names, logos, brands, and trademarks mentioned in this book are the property of their respective owners and are used solely for identification and reference purposes. This book is not endorsed by, directly affiliated with, maintained, authorized, or sponsored by Hugging Face, Inc., Google LLC, Meta Platforms, Inc., NVIDIA Corporation, OpenAI, Inc., or any other trademark holder. The use of any trade name or trademark is for identification and reference purposes only and does not imply any association with the trademark holder.

OPEN-SOURCE ATTRIBUTION: This book references several open-source libraries. The Transformers library (Copyright 2018- The Hugging Face team) is licensed under the Apache License 2.0. The Datasets library is licensed under the Apache License 2.0. The PyTorch framework is licensed under the BSD-style license. All code examples in this book are original works by the author, written for educational purposes. Where code patterns are derived from official open-source documentation, proper attribution has been maintained. Readers are free to use and modify these code examples in their own research and projects. For questions, corrections, or licensing inquiries, contact: rominur@gmail.com | jekardah.com

# Preface

Artificial intelligence research has undergone a remarkable transformation in recent years. What once required expensive hardware, proprietary software, and institutional backing can now be accomplished by a single researcher armed with a web browser and a solid understanding of open-source tools. The barriers to entry have never been lower, and the potential for impactful research has never been higher.

At the center of this shift are two platforms that have democratized AI research: Hugging Face and Google Colab. Hugging Face has become the de facto hub for sharing, discovering, and deploying machine learning models. With over 500,000 models and 100,000 datasets hosted on its platform, it represents the largest open-source machine learning ecosystem in the world. Google Colab has removed the hardware barrier by offering free access to powerful GPUs in a notebook environment that requires zero configuration.

This handbook is the bridge between these two worlds. It is designed to help you, the AI researcher, navigate the Hugging Face ecosystem efficiently while leveraging Google Colab as your primary research environment. Whether you are fine-tuning a language model, building a retrieval-augmented generation pipeline, exploring computer vision models, or processing audio data, this book will show you how to do it effectively and efficiently.

The approach throughout this book is practical and hands-on. Every chapter contains code that you can run directly in a Colab notebook. There are laboratory exercises at the end of major sections that walk you through complete workflows from start to finish. Diagrams, comparison tables, and architectural illustrations are used extensively to help you understand complex concepts visually.

I have deliberately avoided theoretical tangents in favor of actionable knowledge. The goal is not to teach you machine learning from scratch, but to make you productive with the tools that modern AI researchers actually use. The theoretical foundations are important, but they are well-covered by existing textbooks. This handbook focuses on the practical skills that those textbooks often overlook.

I hope this handbook serves as a reliable companion throughout your research journey. The field moves fast, but the fundamentals covered here will remain relevant for years to come. Keep experimenting, keep learning, and keep pushing the boundaries of what is possible.

# Who This Book Is For

This book is written for a diverse audience of practitioners who work with or aspire to work with modern AI tools. The following profiles describe the readers who will benefit most from this handbook:

| Reader Profile | Background | What You Will Gain |
|---|---|---|
| Graduate Student | Basic Python, introductory ML course | Complete workflow from data to deployed model |
| Data Scientist | Statistical modeling, Python, SQL | Deep learning skills using state-of-the-art models |
| ML Engineer | PyTorch/TensorFlow, training pipelines | Efficient use of Hugging Face ecosystem |
| AI Researcher | Published papers, domain expertise | Rapid prototyping and reproducibility techniques |
| Self-taught Developer | Programming experience, ML curiosity | Structured learning path with hands-on labs |

You should have a basic understanding of Python programming and fundamental machine learning concepts such as training, evaluation, overfitting, and gradient descent. Prior experience with deep learning frameworks like PyTorch or TensorFlow is helpful but not strictly required, as the Hugging Face libraries abstract away many of the low-level details.

If you are completely new to programming or machine learning, consider starting with an introductory course first and then returning to this book once you have the fundamentals in place. If you are already experienced, you will find that this book helps you discover efficient workflows, advanced techniques, and best practices that you may not have encountered before.

# How to Use This Book

This handbook is organized into six parts plus appendices, progressing from foundational concepts to advanced applied workflows. You can read it from cover to cover for a comprehensive learning experience, or jump directly to the chapters most relevant to your current needs. Each chapter is designed to be reasonably self-contained.

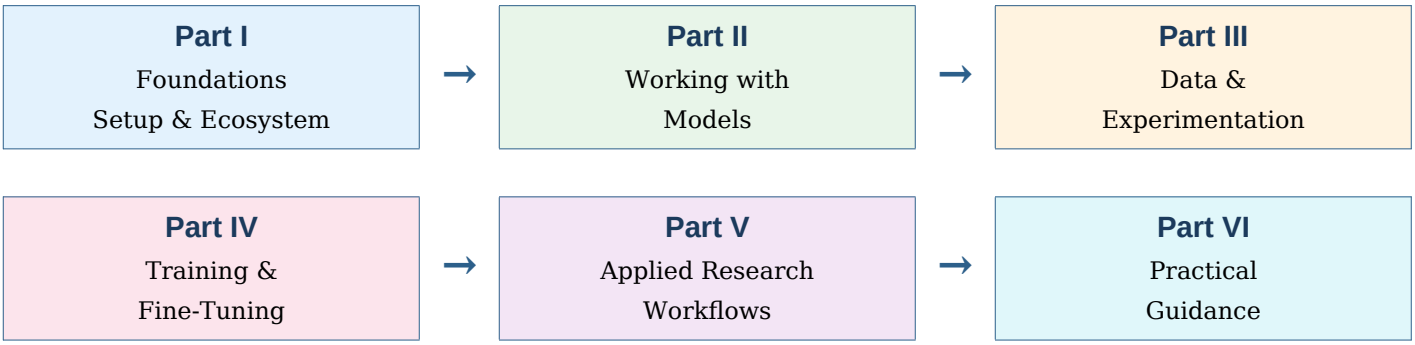


*Figure: Book Structure Overview — Six parts from foundations to practical guidance*

Throughout the book, you will encounter several recurring elements designed to enhance your learning experience:

| Element | Description | Purpose |
|---|---|---|
| Code Blocks | Syntax-highlighted code snippets | Ready to copy into Colab |
| Lab Exercises | Step-by-step guided projects | Hands-on practice |
| Diagrams | Visual architecture and flow charts | Conceptual understanding |
| Comparison Tables | Side-by-side feature analysis | Decision-making support |
| Notes | Additional context and explanations | Deeper understanding |
| Tips | Practical advice and shortcuts | Efficiency improvement |
| Warnings | Common pitfalls and mistakes | Error prevention |

Each chapter builds upon the knowledge from previous chapters, but cross-references are provided whenever a concept from another chapter is required. If you choose to read out of order, use the cross-references to fill in any gaps in your understanding.

The lab exercises are designed to be completed in Google Colab. Each lab includes all the code you need to complete the exercise, along with expected

outputs so you can verify your results. I strongly recommend completing the labs, as hands-on practice is the most effective way to learn these tools.

# Table of Contents

# PART I

## FOUNDATIONS

# Chapter 1. Introduction to Hugging Face and Google Colab

This chapter provides the essential context you need before diving into hands-on work. We will explore what Hugging Face is and why it has become the center of gravity for open-source AI. We will examine why Google Colab is an ideal research environment, and we will compare it with alternative computing options. By the end of this chapter, you will have a clear understanding of the landscape and be ready to set up your research environment in Chapter 2.

## 1.1 What Hugging Face Is

Hugging Face was founded in 2016 as a chatbot startup in New York City. The original product was an AI-powered chatbot aimed at teenagers. However, the company pivoted in 2018 to focus on natural language processing tools, and this pivot turned out to be one of the most consequential decisions in the history of open-source AI.

The release of the Transformers library in late 2018 changed everything. For the first time, researchers and developers had a unified, easy-to-use interface for working with models like BERT, GPT-2, and their variants. The library quickly gained traction, and Hugging Face began building an ecosystem around it.

Today, Hugging Face is best described as the GitHub of machine learning. The platform hosts over 500,000 pre-trained models, more than 100,000 datasets, and thousands of interactive demo applications called Spaces. The company has raised over $395 million in funding and is valued at several billion dollars, reflecting the enormous impact it has had on the AI ecosystem.

The core philosophy of Hugging Face is democratization. The platform makes state-of-the-art AI accessible to anyone with an internet connection. Whether you need a model for text classification, image generation, speech recognition, protein folding, or code generation, chances are that someone has already uploaded a pre-trained version to the Hugging Face Hub.

| Hugging Face Component | Description | Key Feature |
|---|---|---|
| Hub | Central repository for models, datasets, and Spaces | 500K+ models, Git-based versioning |
| Transformers | Unified API for pre-trained models | Auto classes, multi-framework support |

| Hugging Face Component | Description | Key Feature |
|---|---|---|
| Datasets | Efficient data loading and processing | Apache Arrow, streaming, 100K+ datasets |
| Tokenizers | Fast text preprocessing in Rust | BPE, WordPiece, SentencePiece |
| Diffusers | Diffusion model pipelines | Stable Diffusion, ControlNet |
| Evaluate | Standardized metric computation | Accuracy, BLEU, ROUGE, F1 |
| Accelerate | Distributed training abstraction | Multi-GPU, mixed precision |
| PEFT | Parameter-efficient fine-tuning | LoRA, QLoRA, prefix tuning |
| Spaces | Interactive demo hosting | Gradio, Streamlit, Docker |
| Inference API | Serverless model inference | Free tier, production-ready |

*Figure: The Hugging Face Ecosystem — Core components and their roles*

## 1.2 Why Researchers Use Hugging Face

The appeal of Hugging Face for researchers comes down to five key factors that address the most common pain points in AI research:

**Accessibility.** Pre-trained models can be loaded in just two or three lines of code, eliminating weeks of training time. The Auto classes automatically detect the correct model architecture, so you do not need to know the implementation details to get started. This dramatically lowers the barrier to entry for researchers who want to build on existing work.

**Community.** The community contributes thousands of new models every month, ensuring that the latest research is always within reach. Discussion forums, model cards, and community contributions create a collaborative environment that accelerates progress. When a new paper is published, community members often have an implementation available on the Hub within days.

**Speed.** The unified API means that switching between different model architectures requires minimal code changes. You can experiment with dozens of models in a single afternoon, comparing their performance on your specific task. This rapid iteration cycle is essential for productive research.

**Reproducibility.** Every model on the Hub comes with a model card documenting its architecture, training data, intended use cases, and limitations. Git-based versioning ensures that you can always retrieve the exact version of a model. This level of transparency is essential for scientific rigor.

**Framework Flexibility.** Hugging Face integrates with PyTorch, TensorFlow, and JAX. Models can be converted between frameworks, and the API remains consistent regardless of the backend. This means you can incorporate Hugging Face into your existing research pipeline without significant refactoring.

## 1.3 Why Google Colab Matters

Google Colab, short for Google Colaboratory, is a free cloud-based Jupyter notebook environment provided by Google Research. It allows you to write and execute Python code in your browser without any local setup. More importantly, it provides free access to GPU and TPU hardware, which is essential for training and running modern machine learning models.

For researchers who do not have access to institutional computing resources, Colab is a lifeline. A standard free Colab session provides access to an NVIDIA T4 GPU with approximately 15 GB of VRAM and around 12.7 GB of system RAM. This is sufficient for many fine-tuning tasks and most inference workloads. Higher tiers offer more powerful hardware and longer session times.

| Feature | Free Tier | Colab Pro | Colab Pro+ |
|---|---|---|---|
| GPU Types | T4 | T4, V100, A100 | T4, V100, A100 (priority) |
| GPU Memory | ~15 GB | Up to 40 GB | Up to 80 GB |
| System RAM | ~12.7 GB | Up to 32 GB | Up to 52 GB |
| Max Session | ~12 hours | ~24 hours | ~24 hours |
| Idle Timeout | ~90 minutes | ~90 minutes | ~90 minutes (background) |
| Background Execution | No | No | Yes |
| Disk Space | ~78 GB | ~225 GB | ~225 GB |
| Price | Free | ~$12/month | ~$50/month |

*Figure: Google Colab Tier Comparison — Hardware and features by subscription level*

The notebook format encourages experimentation. You can mix code, text, visualizations, and interactive widgets in a single document, making it easy to document your thought process as you work. Notebooks are inherently shareable, which aligns with the open science principles that many researchers champion.

Perhaps most importantly, Colab eliminates environment management headaches. There is no need to install CUDA drivers, configure virtual environments, or deal with dependency conflicts on your local machine. The pre-installed libraries cover most common use cases, and any additional packages can be installed with a single pip command.

## 1.4 Colab vs. Local Workstation vs. Cloud VM

Choosing the right computing environment depends on your specific needs, budget, and the scale of your experiments. Each option has distinct advantages and limitations that are worth understanding before committing to a workflow.

| Criterion | Google Colab | Local Workstation | Cloud VM (AWS/GCP/Azure) |
|---|---|---|---|
| Setup Time | Zero (browser-based) | Hours to days | 30–60 minutes |
| Cost | Free – $50/month | $2,000–$10,000+ upfront | $0.50–$30+/hour |
| GPU Access | Shared, not guaranteed | Dedicated, always available | On-demand, configurable |
| Session Persistence | Ephemeral (lost on timeout) | Permanent | Persistent (runs until stopped) |
| Max GPU Memory | Up to 80 GB (Pro+) | Limited by hardware budget | Up to 80 GB+ (A100, H100) |
| Multi-GPU | Not available | Possible with hardware | Easily configurable |
| Data Storage | Google Drive / temp | Local SSD/HDD | Cloud storage (S3, GCS) |
| Collaboration | Easy sharing via link | Requires setup (Git, SSH) | Requires setup |
| Maintenance | None (Google manages) | Full responsibility | Shared responsibility |
| Best For | Prototyping, learning, small | Long-running training, full control | Large-scale training, production |

| Criterion | Google Colab | Local Workstation | Cloud VM (AWS/GCP/Azure) |
|---|---|---|---|
| | experiments | | |

*Figure: Computing Environment Comparison — Detailed feature matrix*

For most of the workflows covered in this book, Google Colab will be more than sufficient. When we encounter tasks that push Colab's limits, we will discuss specific strategies for working within those constraints. Chapter 14 covers techniques like quantization and mixed precision that extend what is possible on limited hardware, and Chapter 22 discusses when and how to transition to more powerful environments.

## The Hugging Face + Colab Research Workflow

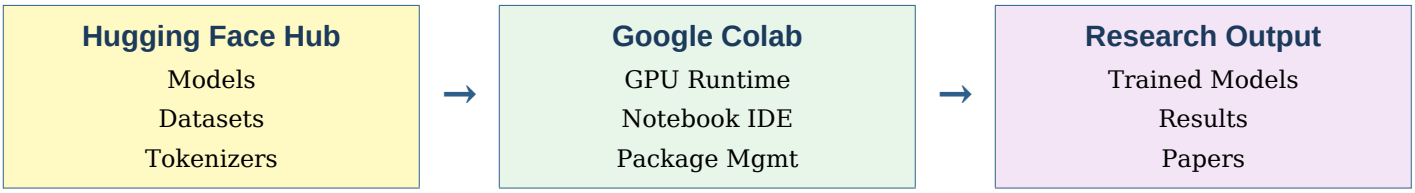


*Figure: High-level research workflow using Hugging Face and Google Colab*

### 🔬 LAB: Exploring the Hugging Face Hub

In this lab, you will explore the Hugging Face Hub to familiarize yourself with its interface and content. You will search for models, examine model cards, and understand the information available for each model.

**Step 1:** Open your browser and navigate to huggingface.co. Create an account if you do not already have one. Take note of the main navigation elements: Models, Datasets, Spaces, and Documentation.

**Step 2:** Click on Models and use the task filter to find models for "Text Classification". Sort by "Most Downloads" and examine the top 5 results. For each model, note: the model name, number of downloads, the framework, and the license.

**Step 3:** Click on the most downloaded text classification model and read its model card. Identify: the base architecture, the training dataset, the reported performance metrics, and any listed limitations.

**Step 4:** Navigate to Datasets and search for "imdb". Open the IMDB dataset card and explore its structure. Note the number of examples in each split, the features, and the dataset size.

**Step 5:** Explore the Spaces section. Find a text generation demo and try it out. Observe the response time and quality of the outputs. Consider how you might create a similar demo for your own research.

| Item | Your Observations |
|---|---|
| Top text classification model name | (Fill in) |
| Downloads count | (Fill in) |
| Base architecture | (Fill in) |
| Training dataset | (Fill in) |
| Reported accuracy | (Fill in) |
| License type | (Fill in) |
| IMDB dataset size (train split) | (Fill in) |

*Figure: Lab 1 Observation Sheet — Record your findings*

> ✓ **Tip:** *Bookmark the models and datasets you find most relevant to your research. The Hugging Face Hub allows you to "like" items, which creates a personal collection that you can return to later.*

## Understanding the Transformer Revolution

The transformer architecture, introduced in the landmark 2017 paper 'Attention Is All You Need' by Vaswani et al., fundamentally changed the landscape of artificial intelligence. Before transformers, recurrent neural networks (RNNs) and long short-term memory networks (LSTMs) were the dominant architectures for sequential data processing. These architectures processed data one step at a time, which made training slow and limited the ability to capture long-range dependencies in text.

Transformers solved these problems with the self-attention mechanism, which allows the model to look at all positions in the input simultaneously. This parallelism dramatically speeds up training and enables the model to learn relationships between distant words in a text. The result was a new generation of models that could be trained on vastly larger datasets in less time, leading to the remarkable capabilities we see today in models like BERT, GPT, T5, and their descendants.

The impact on research has been transformative. Tasks that previously required specialized architectures and extensive engineering can now be addressed with a single, general-purpose model architecture. Fine-tuning a pre-trained transformer on a specific task often outperforms task-specific models that took years to develop. This is the power of transfer learning at scale, and it is the fundamental insight that makes Hugging Face so valuable to researchers.

## The Open-Source AI Movement

Hugging Face sits at the center of the open-source AI movement, which represents a philosophical commitment to making AI technology accessible and transparent. In contrast to closed-source approaches where models are kept proprietary and accessed only through APIs, the open-source movement believes that AI development benefits from community participation, peer review, and shared progress.

This philosophy has practical implications for researchers. Open-source models can be inspected, modified, and adapted for specific use cases. Their training data and methodologies are documented, enabling reproducible research. And the collaborative nature of open-source development means that improvements are shared across the entire community, accelerating progress for everyone.

The numbers speak for themselves: Hugging Face has over 50,000 organizations on its platform, including major tech companies, academic institutions, and independent research labs. The platform sees millions of model downloads per day, and new models are uploaded at a rate of hundreds per day. This ecosystem effect creates a virtuous cycle: more users attract more contributors, which creates more value for all users.

| Year | Milestone | Impact |
|---|---|---|
| 2017 | Transformer paper published | New architecture paradigm |
| 2018 | BERT released by Google | Pre-training revolution for NLP |
| 2018 | Hugging Face Transformers library launched | Democratized access to BERT |
| 2019 | GPT-2 released by OpenAI | Large-scale text generation |
| 2020 | Hugging Face Hub launched | Centralized model sharing |
| 2020 | ViT (Vision Transformer) published | Transformers for computer vision |
| 2021 | DALL-E and image generation models | Text-to-image capabilities |
| 2022 | ChatGPT launched | Public awareness of LLMs |
| 2022 | Stable Diffusion open-sourced | Open image generation |
| 2023 | LLaMA released by Meta | Open-source large language models |
| 2023 | Mistral 7B released | Efficient open-source LLMs |
| 2024 | Mixture-of-Experts models proliferate | Efficient scaling |
| 2025 | Multi-modal open models mature | Vision, audio, and text unified |

*Figure: Timeline of Major AI Milestones Relevant to Hugging Face*

## Colab's Role in Democratizing AI Research

Google Colab was launched in 2017 and initially targeted at machine learning education and research. Its key innovation was combining the familiar Jupyter notebook interface with free cloud computing resources, including GPUs. This combination removed the two biggest barriers to AI research: the need for expensive hardware and the complexity of environment setup.

Before Colab, a researcher wanting to train a neural network needed either a personal workstation with a GPU (costing thousands of dollars) or access to institutional computing clusters (often requiring approval processes and quotas). Colab eliminated both barriers with a single URL: colab.research.google.com.

The impact on global AI research has been profound. Researchers in developing countries, students at institutions without GPU clusters, and independent researchers all gained access to the same computing resources that were previously available only at well-funded labs. This democratization has led to more diverse perspectives in AI research and more rapid progress across the field.

Colab's integration with Google Drive provides a natural solution for data persistence and collaboration. Notebooks can be shared via links, making it trivial to distribute code alongside research papers. The ability to collaborate on notebooks in real-time, similar to Google Docs, further supports team-based research workflows.

## The Hugging Face and Colab Synergy

Hugging Face and Google Colab complement each other in ways that make them greater than the sum of their parts. Hugging Face provides the models, datasets, and tools, while Colab provides the computing environment to use them. Together, they enable a complete research workflow from data exploration to model deployment without any local infrastructure.

This synergy is evident in the number of tutorials, courses, and research papers that use both platforms together. The Hugging Face course, which has been taken by hundreds of thousands of learners, uses Google Colab as its primary computing environment. Many research papers include Colab notebooks as supplementary materials, making it easy for readers to reproduce the results.

Throughout this book, we will leverage this synergy to its fullest extent. Every code example is designed to run in Colab, and every workflow assumes that you have access to the Hugging Face Hub. By the time you finish this book, you will be able to use these two platforms together as naturally as using a web browser and a search engine.

## A Day in the Life of an AI Researcher Using Hugging Face and Colab

To make the abstract concrete, let us walk through a typical research day that uses both platforms:

| Time | Activity | Platform Used | Details |
|---|---|---|---|
| 9:00 AM | Read latest papers on arXiv | Browser | Identify models to evaluate |
| 9:30 AM | Search for models on Hugging Face | HF Hub | Compare model cards and metrics |
| 10:00 AM | Open Colab notebook | Colab | Load model with pipeline API |
| 10:30 AM | Run initial experiments | Colab + HF | Test model on sample data |
| 11:00 AM | Load and preprocess dataset | Colab + HF Datasets | Tokenize, split, prepare |

| Time | Activity | Platform Used | Details |
|---|---|---|---|
| 12:00 PM | Configure fine-tuning | Colab + HF Trainer | Set hyperparameters |
| 12:30 PM | Start training | Colab (GPU) | Monitor loss and metrics |
| 2:00 PM | Evaluate results | Colab + HF Evaluate | Compute metrics, error analysis |
| 3:00 PM | Save checkpoint to Drive | Colab + Google Drive | Persist model weights |
| 3:30 PM | Push model to Hub | Colab + HF Hub | Share with community |
| 4:00 PM | Build Gradio demo | Colab + Gradio | Interactive showcase |
| 4:30 PM | Deploy to Spaces | HF Spaces | Public demo URL |
| 5:00 PM | Document findings | Colab (Markdown) | Update notebook narrative |

*Figure: A typical research day using Hugging Face and Google Colab*

# Chapter 2. Setting Up the Research Environment

A properly configured research environment is the foundation of productive AI research. This chapter walks you through every aspect of setting up Google Colab for Hugging Face work, from basic notebook operations to advanced configuration techniques. By the end of this chapter, you will have a fully functional environment ready for the hands-on work in subsequent chapters.

## 2.1 Colab Basics

When you open Google Colab for the first time, you are presented with a clean notebook interface. The notebook consists of two types of cells: code cells that execute Python code, and text cells that accept Markdown formatting for documentation and notes. Understanding how to work efficiently with these cells is the first step toward productive research.

To create a new notebook, navigate to colab.research.google.com and click New Notebook. The interface will look familiar if you have used Jupyter notebooks before. The key difference is that Colab runs entirely in the cloud, so your code executes on Google's servers rather than on your local machine. This means that your notebook's state, including all variables, loaded models, and installed packages, exists only for the duration of your session.

Colab sessions have a maximum duration that depends on your subscription tier. Free users get sessions that last up to 12 hours, with an idle timeout of

approximately 90 minutes. Understanding these limits is important for planning your research workflow, especially for long training runs.

### Notebook Interface Anatomy

| Interface Element | Location | Function |
|---|---|---|
| Menu Bar | Top of page | File, Edit, View, Insert, Runtime, Tools, Help |
| Code Cell | Main area | Write and execute Python code |
| Text Cell | Main area | Markdown documentation and notes |
| + Code / + Text | Below cells or toolbar | Insert new cells |
| Runtime Info | Top-right corner | RAM/Disk usage, connected status |
| File Browser | Left sidebar | Browse files in the runtime |
| Table of Contents | Left sidebar | Navigate by headings |
| Search/Replace | Ctrl+H | Find and replace across notebook |

*Figure: Colab Interface Elements — Key components and their locations*

## 2.2 Using the GPU

GPU acceleration is essential for most machine learning tasks. Without a GPU, operations that take seconds could take hours. To enable GPU acceleration in Colab, go to Runtime in the menu bar, then select Change runtime type. In the dialog that appears, set the Hardware accelerator to GPU.

You can verify that a GPU is available and check its specifications using the following code:

```
# Check GPU availability and specifications
import torch

print(f'PyTorch version: {torch.__version__}')
print(f'CUDA available: {torch.cuda.is_available()}')

if torch.cuda.is_available():
    print(f'GPU device: {torch.cuda.get_device_name(0)}')
```

```
    print(f'GPU memory: {torch.cuda.get_device_properties(0).total_mem / 1e9:.1f} GB')
    print(f'CUDA version: {torch.version.cuda}')
else:
    print('No GPU available. Check Runtime > Change runtime type.')
```

> ℹ️ **Note:** *GPU availability is not guaranteed on the free tier. During peak usage periods, you may be placed in a queue or temporarily restricted. If you consistently need guaranteed GPU access, consider upgrading to Colab Pro.*

| GPU Model | VRAM | Compute Capability | Typical Tier | Best For |
|---|---|---|---|---|
| NVIDIA T4 | 15 GB | 7.5 (Turing) | Free / Pro | Inference, small fine-tuning |
| NVIDIA V100 | 16 GB | 7.0 (Volta) | Pro | Medium fine-tuning, training |
| NVIDIA A100 | 40/80 GB | 8.0 (Ampere) | Pro+ | Large model fine-tuning |
| NVIDIA L4 | 24 GB | 8.9 (Ada) | Pro | Efficient inference, medium training |

*Figure: GPU Models Available in Google Colab*

## 2.3 Installing Packages

While Colab comes with many pre-installed libraries, you will need to install additional packages for Hugging Face workflows. The standard approach is to use pip install commands in code cells at the top of your notebook.

```
# Install core Hugging Face packages
!pip install -q transformers datasets tokenizers evaluate
!pip install -q accelerate peft bitsandbytes
!pip install -q sentence-transformers faiss-cpu
!pip install -q gradio huggingface_hub
```

The -q flag suppresses verbose output, keeping your notebook clean. It is good practice to install all required packages at the very beginning of your notebook so that dependencies are resolved before any code runs.

> ⚠️ **Warning:** *Some installations require a runtime restart. Colab will display a 'Restart Runtime' button when this happens. After restarting, you need to re-run all cells from the beginning, including the pip install commands.*

| Package | Purpose | When You Need It |
|---|---|---|
| transformers | Model loading, inference, training | Every Hugging Face project |
| datasets | Data loading and processing | Any data-dependent workflow |
| tokenizers | Fast text tokenization | Installed with transformers |
| evaluate | Metric computation | Model evaluation |
| accelerate | Distributed training helpers | Training and fine-tuning |
| peft | Parameter-efficient fine-tuning | LoRA, QLoRA fine-tuning |
| bitsandbytes | Quantization support | Running large models |
| sentence-transformers | Sentence embeddings | Semantic search, RAG |
| faiss-cpu | Vector similarity search | Retrieval systems |
| gradio | Interactive demos | Building and sharing demos |

*Figure: Essential Packages for Hugging Face Research in Colab*

## 2.4 Connecting Google Drive

Google Drive integration is one of Colab's most valuable features for researchers. It allows you to persist files across sessions, access large datasets without re-downloading them, and save model checkpoints and results that survive session disconnections.

```
# Mount Google Drive
from google.colab import drive
drive.mount('/content/drive')

# Create a project directory
import os
project_dir = '/content/drive/MyDrive/hf_research'
os.makedirs(project_dir, exist_ok=True)

# Verify the mount
print(f'Project directory: {project_dir}')
```

```
print(f'Files: {os.listdir(project_dir)}')
```

Once mounted, your Drive appears as a directory at /content/drive/MyDrive. You can read from and write to this directory just like any other file system path. This is particularly useful for saving trained model weights, which can be many gigabytes in size and time-consuming to regenerate.

### Recommended Google Drive Project Structure

| Directory | Contents | Example Files |
|---|---|---|
| /hf_research/ | Project root | README.md |
| /hf_research/data/ | Datasets and raw data | train.csv, val.csv |
| /hf_research/models/ | Saved model checkpoints | checkpoint-1000/ |
| /hf_research/results/ | Evaluation results and logs | eval_results.json |
| /hf_research/notebooks/ | Backed up notebooks | experiment_01.ipynb |
| /hf_research/configs/ | Configuration files | training_config.yaml |
| /hf_research/cache/ | HF model/dataset cache | transformers/ |

*Figure: Recommended project directory structure in Google Drive*

> ✓ **Tip:** *Set the HF_HOME environment variable to point to your Drive cache directory. This ensures that downloaded models and datasets persist across sessions: os.environ['HF_HOME'] = '/content/drive/MyDrive/hf_research/cache'*

## 2.5 Tokens and Authentication

To access certain models and datasets on the Hugging Face Hub, you need to authenticate with a Hugging Face access token. Some models, particularly those from Meta (LLaMA), Google (Gemma), and Mistral AI, require you to accept a license agreement before you can download them.

```
# Method 1: Interactive login (recommended for Colab)
from huggingface_hub import login
login()  # Prompts for token input

# Method 2: Login with token directly
```

```
# Only use this in private notebooks!
login(token='hf_your_token_here')

# Method 3: Using environment variable
import os
os.environ['HF_TOKEN'] = 'hf_your_token_here'
```

> ⚠️ **Warning:** *Never hard-code your token in a notebook that you plan to share publicly. Use the interactive login method or store your token in a Colab secret (Settings > Secrets) and retrieve it with google.colab.userdata.get('HF_TOKEN').*

| Token Type | Permissions | Use Case |
|---|---|---|
| Read (fine-grained) | Download public and gated models/datasets | Most research workflows |
| Write (fine-grained) | Read + upload models/datasets/spaces | Publishing to the Hub |
| Read (legacy) | Basic read access | Simple inference only |
| Write (legacy) | Full read/write access | Legacy workflows |

*Figure: Hugging Face Token Types and Their Permissions*

## 🔬 LAB: Complete Environment Setup and Verification

In this lab, you will set up a complete research environment in Google Colab and verify that every component is working correctly. This notebook will serve as a template that you can reuse for future projects.

**Step 1: Create a new Colab notebook.** Go to colab.research.google.com and create a new notebook. Rename it to 'HF_Environment_Setup.ipynb' by clicking on the title at the top.

**Step 2: Enable GPU.** Go to Runtime > Change runtime type > GPU. Click Save.

**Step 3: Install packages.** Create a code cell and run the following:

```
# Cell 1: Install all required packages
!pip install -q transformers datasets tokenizers evaluate
```

```
!pip install -q accelerate peft bitsandbytes
!pip install -q sentence-transformers
print('All packages installed successfully!')
```

**Step 4: Verify GPU and library versions.** Create a new code cell:

```
# Cell 2: Environment verification
import torch
import transformers
import datasets

print('=== Environment Report ===')
print(f'Python:       {__import__("sys").version.split()[0]}')
print(f'PyTorch:      {torch.__version__}')
print(f'Transformers: {transformers.__version__}')
print(f'Datasets:     {datasets.__version__}')
print(f'CUDA:         {torch.cuda.is_available()}')
if torch.cuda.is_available():
    print(f'GPU:          {torch.cuda.get_device_name(0)}')
    mem = torch.cuda.get_device_properties(0).total_mem
    print(f'GPU Memory:   {mem / 1e9:.1f} GB')
print('=== Setup Complete ===')
```

**Step 5: Mount Google Drive and create project structure.** Create a new code cell:

```
# Cell 3: Mount Drive and create project structure
from google.colab import drive
drive.mount('/content/drive')

import os
base = '/content/drive/MyDrive/hf_research'
for subdir in ['data', 'models', 'results', 'notebooks', 'cache']:
    os.makedirs(f'{base}/{subdir}', exist_ok=True)
    print(f'Created: {base}/{subdir}')

# Set HF cache to Drive for persistence
os.environ['HF_HOME'] = f'{base}/cache'
```

**Step 6: Authenticate with Hugging Face.** Create a new code cell:

```
# Cell 4: Hugging Face authentication
```

```
from huggingface_hub import login
login()  # Enter your token when prompted
```

**Step 7: Test a quick inference.** Verify everything works end-to-end:

```
# Cell 5: Quick inference test
from transformers import pipeline

classifier = pipeline('sentiment-analysis', device=0)
results = classifier([
    'I love working with Hugging Face!',
    'This setup process was really frustrating.',
    'The documentation could be better but overall good.',
])

for text, result in zip(texts, results):
    print(f'{result["label"]} ({result["score"]:.4f}): {text}')
print('\nEnvironment setup verified successfully!')
```

**Expected Output:** You should see three sentiment classifications with confidence scores. If any step fails, review the error message and refer to the troubleshooting section in Chapter 20.

## Colab Secrets Management

Google Colab provides a built-in secrets management feature that allows you to store sensitive information like API keys and tokens without exposing them in your notebook code. This is the recommended approach for managing Hugging Face tokens, API keys, and other credentials.

```
# Using Colab Secrets (recommended for tokens)
from google.colab import userdata

# Store your token in Colab Settings > Secrets
# Name: HF_TOKEN, Value: hf_your_token_here
hf_token = userdata.get('HF_TOKEN')

# Use it for authentication
from huggingface_hub import login
login(token=hf_token)
print('Authenticated successfully!')
```

## Understanding Runtime Resources

Knowing how to monitor and manage your Colab runtime resources is crucial for efficient research. Running out of RAM or disk space can cause your session to crash, potentially losing unsaved work. Here are the essential monitoring commands:

```
# Complete resource monitoring toolkit
import psutil, torch, os, shutil

def resource_report():
    # CPU and RAM
    ram = psutil.virtual_memory()
    print(f'=== System Resources ===')
    print(f'CPU cores:  {psutil.cpu_count()}')
    print(f'RAM used:   {ram.used / 1e9:.1f} / {ram.total / 1e9:.1f} GB
({ram.percent}%)')

    # Disk
    disk = shutil.disk_usage('/')
    print(f'Disk used:  {disk.used / 1e9:.1f} / {disk.total / 1e9:.1f}
GB')
```

```
    # GPU
    if torch.cuda.is_available():
        gpu_mem = torch.cuda.get_device_properties(0).total_mem
        gpu_alloc = torch.cuda.memory_allocated(0)
        gpu_reserved = torch.cuda.memory_reserved(0)
        print(f'\n=== GPU Resources ===')
        print(f'GPU:        {torch.cuda.get_device_name(0)}')
        print(f'VRAM total: {gpu_mem / 1e9:.1f} GB')
        print(f'VRAM alloc: {gpu_alloc / 1e9:.2f} GB')
        print(f'VRAM cache: {gpu_reserved / 1e9:.2f} GB')
    else:
        print('\nNo GPU available!')

resource_report()
```

> ✓ **Tip:** *Run resource_report() before and after loading large models to understand their memory footprint. This helps you plan batch sizes and decide whether quantization is necessary.*

## Optimizing Colab for Long Experiments

Long training runs in Colab require special precautions. Sessions can be disconnected due to idle timeouts, resource limits, or server-side maintenance. The following strategies will help you protect your work:

| Strategy | Implementation | Protects Against |
|---|---|---|
| Auto-save to Drive | Point output_dir to Drive path | Session disconnection |
| Frequent checkpoints | save_strategy='steps', save_steps=500 | Mid-training crashes |
| Resume from checkpoint | trainer.train(resume_from_checkpoint=True) | Interrupted training |
| Keep session alive | Interact periodically or use Colab Pro | Idle timeout |
| Pin library versions | !pip install transformers==4.x.x | Environment changes |
| Cache models to Drive | os.environ['HF_HOME'] = Drive path | Re-downloads |
| Use gradient | gradient_accumulation_steps= | Memory limits |

| Strategy | Implementation | Protects Against |
|---|---|---|
| accumulation | 4 | |
| Enable mixed precision | fp16=True in TrainingArguments | Speed + memory |

*Figure: Strategies for Protecting Long Experiments in Colab*

## Setting Up a Reusable Template Notebook

Creating a template notebook that you copy for each new experiment ensures consistency and saves time. Here is a recommended structure for your template:

```
# ============================================
# EXPERIMENT TEMPLATE - Copy for each experiment
# ============================================
# Experiment: [NAME]
# Date: [DATE]
# Author: [YOUR NAME]
# Description: [WHAT YOU'RE TESTING]
# Hypothesis: [WHAT YOU EXPECT]
# ============================================

# --- Cell 1: Environment Setup ---
!pip install -q transformers datasets evaluate accelerate peft

# --- Cell 2: Configuration ---
CONFIG = {
    'model_name': 'distilbert-base-uncased',
    'dataset': 'imdb',
    'learning_rate': 2e-5,
    'batch_size': 16,
    'epochs': 3,
    'max_length': 256,
    'seed': 42,
    'output_dir': '/content/drive/MyDrive/hf_research/exp_XXX',
}

# --- Cell 3: Seeds and Imports ---
from transformers import set_seed
```

```
set_seed(CONFIG['seed'])

# --- Cell 4: Mount Drive ---
from google.colab import drive
drive.mount('/content/drive')
import os
os.makedirs(CONFIG['output_dir'], exist_ok=True)

# --- Cell 5: Data Loading ---
# [Your data loading code]

# --- Cell 6: Model Loading ---
# [Your model loading code]

# --- Cell 7: Training ---
# [Your training code]

# --- Cell 8: Evaluation ---
# [Your evaluation code]

# --- Cell 9: Save Results ---
import json
with open(f'{CONFIG["output_dir"]}/config.json', 'w') as f:
    json.dump(CONFIG, f, indent=2)
print('Experiment complete! Results saved to:', CONFIG['output_dir'])
```

# Chapter 3. Understanding the Hugging Face Ecosystem

The Hugging Face ecosystem consists of several interconnected libraries, each serving a specific purpose in the machine learning workflow. Understanding how these libraries work together is essential for efficient research. This chapter provides a detailed overview of each major component, with practical examples showing how they integrate with each other and with Google Colab.

## 3.1 Ecosystem Architecture

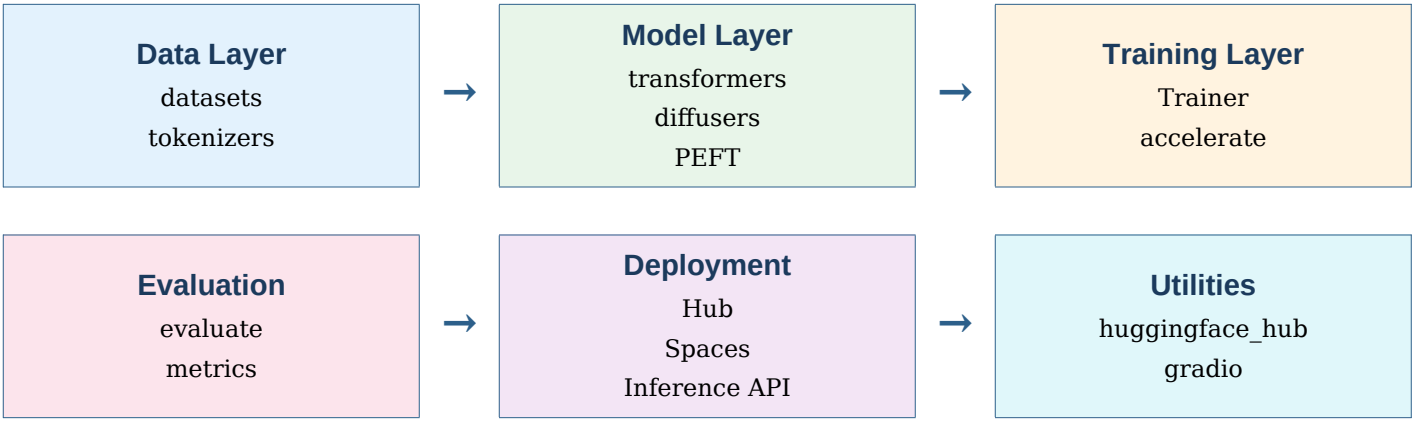


*Figure: Hugging Face Ecosystem Architecture — Libraries organized by function*

## 3.2 Transformers Library

The Transformers library is the flagship product of Hugging Face and the most widely used library in the ecosystem. It provides a unified API for working with over 200 model architectures across text, image, audio, and multimodal modalities. The library's design philosophy centers on simplicity and flexibility, making it easy to get started while also supporting advanced customization.

At its core, the library offers two main abstractions. Model classes give you fine-grained control over how models are loaded, configured, and used. Pipeline functions provide a high-level API that handles tokenization, inference, and post-processing in a single function call. Most researchers start with pipelines and move to model classes when they need more control.

The Auto classes are particularly powerful. AutoModel, AutoTokenizer, AutoConfig, and their task-specific variants like AutoModelForSequenceClassification automatically detect the correct architecture based on the model name or path. This means you can write code that works with any model on the Hub without knowing the specific implementation.

| Auto Class | Purpose | Example |
|---|---|---|
| AutoModel | Base model (no task head) | Feature extraction, custom architectures |
| AutoModelForSequenceClassification | Classification head | Sentiment analysis, topic classification |

| Auto Class | Purpose | Example |
|---|---|---|
| AutoModelForTokenClassification | Per-token classification | Named entity recognition, POS tagging |
| AutoModelForQuestionAnswering | QA head | Extractive question answering |
| AutoModelForCausalLM | Autoregressive generation | Text generation, chatbots |
| AutoModelForSeq2SeqLM | Encoder-decoder generation | Translation, summarization |
| AutoModelForImageClassification | Image classification head | Image labeling, categorization |
| AutoModelForObjectDetection | Object detection head | Bounding box prediction |
| AutoModelForSpeechSeq2Seq | Speech-to-text | Transcription, ASR |
| AutoTokenizer | Matching tokenizer | All text-based tasks |

*Figure: Common Auto Classes in the Transformers Library*

```
# Loading models with Auto classes
from transformers import AutoModelForSequenceClassification, AutoTokenizer

# These two lines work with ANY classification model
model_name = 'distilbert-base-uncased-finetuned-sst-2-english'
tokenizer = AutoTokenizer.from_pretrained(model_name)
model = AutoModelForSequenceClassification.from_pretrained(model_name)

# Swap to a completely different architecture with no code changes
model_name = 'roberta-base'  # Different architecture, same API
tokenizer = AutoTokenizer.from_pretrained(model_name)
model = AutoModelForSequenceClassification.from_pretrained(model_name)
```

## 3.3 Datasets Library

The Datasets library is designed for efficient loading and processing of large-scale datasets. It uses Apache Arrow as its in-memory format, which enables memory-mapped access to data that is too large to fit in RAM. This is particularly important in the memory-constrained Colab environment.

Loading a dataset from the Hub is a one-liner. The load_dataset function accepts a dataset name and returns a DatasetDict object containing train,

validation, and test splits. You can also load specific configurations, subsets, streaming versions, or local files in formats like CSV, JSON, and Parquet.

## Dataset Processing Pipeline

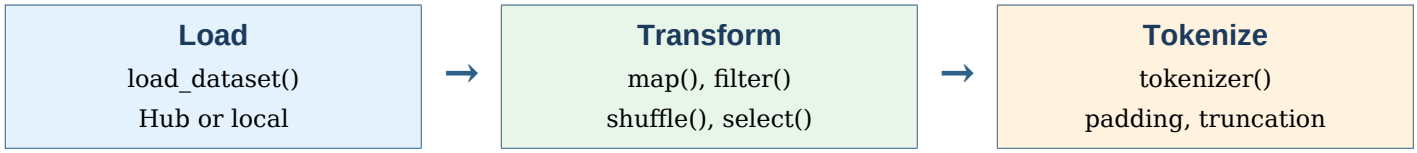


*Figure: Typical dataset processing pipeline in Hugging Face*

```
from datasets import load_dataset

# Load from Hub
dataset = load_dataset('imdb')
print(dataset)
# DatasetDict({
#     train: Dataset({features: ['text', 'label'], num_rows: 25000})
#     test: Dataset({features: ['text', 'label'], num_rows: 25000})
# })

# Load specific split
train_data = load_dataset('imdb', split='train')

# Load with streaming for large datasets
stream = load_dataset('c4', 'en', split='train', streaming=True)

# Load from local files
local_data = load_dataset('csv', data_files='my_data.csv')
```

| Operation | Method | Description | Memory Impact |
|---|---|---|---|
| Load | load_dataset() | Download and cache dataset | Full dataset in Arrow format |
| Filter | .filter(fn) | Keep rows matching condition | Returns new dataset view |
| Map | .map(fn) | Transform each example | Can add/modify columns |
| Select | .select(indices) | Choose specific rows | Returns subset view |

| Operation | Method | Description | Memory Impact |
|---|---|---|---|
| Shuffle | .shuffle(seed) | Randomize row order | Returns shuffled view |
| Sort | .sort(column) | Order by column value | Returns sorted view |
| Rename | .rename_column() | Rename a column | Metadata change only |
| Remove | .remove_columns() | Drop columns | Reduces memory |
| Streaming | streaming=True | Lazy loading, no full download | Minimal memory |

*Figure: Dataset Operations — Methods, descriptions, and memory implications*

## 3.4 Tokenizers Library

The Tokenizers library provides fast implementations of the most commonly used tokenization algorithms. Written in Rust with Python bindings, it is orders of magnitude faster than pure Python implementations. Tokenization is the critical bridge between raw text and numerical representations that models can process.

| Algorithm | Used By | Approach | Vocabulary Size |
|---|---|---|---|
| WordPiece | BERT, DistilBERT, ELECTRA | Greedy subword splitting | ~30,000 |
| Byte-Pair Encoding (BPE) | GPT-2, RoBERTa, LLaMA | Frequency-based merging | ~50,000 |
| SentencePiece (Unigram) | T5, ALBERT, XLNet | Probabilistic subword model | ~32,000 |
| Byte-level BPE | GPT-2, Mistral | BPE on raw bytes | ~50,000+ |

*Figure: Common Tokenization Algorithms and the Models That Use Them*

```
from transformers import AutoTokenizer

tokenizer = AutoTokenizer.from_pretrained('bert-base-uncased')
text = 'Hugging Face transforms AI research'

# Encode text to token IDs
tokens = tokenizer(text, return_tensors='pt')
```

```
print(f'Input IDs: {tokens["input_ids"]}')
print(f'Attention mask: {tokens["attention_mask"]}')

# Decode back to text
decoded = tokenizer.decode(tokens['input_ids'][0])
print(f'Decoded: {decoded}')

# View individual tokens
token_list = tokenizer.tokenize(text)
print(f'Tokens: {token_list}')
```

## 3.5 Evaluate Library

The Evaluate library provides a standardized way to compute metrics for model evaluation. Consistency in evaluation is crucial for reproducible research. By using a standardized library, you ensure that your results are directly comparable with those reported in published papers and model cards.

```
import evaluate

# Load and compute a single metric
accuracy = evaluate.load('accuracy')
result = accuracy.compute(
    predictions=[1, 0, 1, 1, 0],
    references=[1, 0, 1, 0, 0]
)
print(f'Accuracy: {result["accuracy"]:.4f}')  # 0.8000

# Combine multiple metrics
clf_metrics = evaluate.combine(['accuracy', 'f1', 'precision', 'recall'])
results = clf_metrics.compute(
    predictions=[1, 0, 1, 1, 0],
    references=[1, 0, 1, 0, 0]
)
for metric, value in results.items():
    print(f'{metric}: {value:.4f}')
```

| Metric | Task | Range | Higher = Better? |
|---|---|---|---|
| Accuracy | Classification | 0–1 | Yes |
| F1 Score | Classification | 0–1 | Yes |
| Precision | Classification | 0–1 | Yes |
| Recall | Classification | 0–1 | Yes |
| BLEU | Translation / Generation | 0–1 | Yes |
| ROUGE | Summarization | 0–1 | Yes |
| Perplexity | Language Modeling | 1–∞ | No (lower is better) |
| WER | Speech Recognition | 0–∞ | No (lower is better) |
| Exact Match | Question Answering | 0–1 | Yes |

*Figure: Common Evaluation Metrics by Task*

## 3.6 Diffusers Library

The Diffusers library is Hugging Face's toolkit for diffusion models, which power modern image generation systems like Stable Diffusion, DALL-E, and Midjourney-style models. The library provides pre-built pipelines for text-to-image generation, image-to-image transformation, inpainting, super-resolution, and video generation.

For researchers interested in generative AI, Diffusers offers a clean and modular architecture. The library separates the noise scheduler, the model backbone (typically a U-Net or transformer), and the inference pipeline into distinct components that can be customized independently.

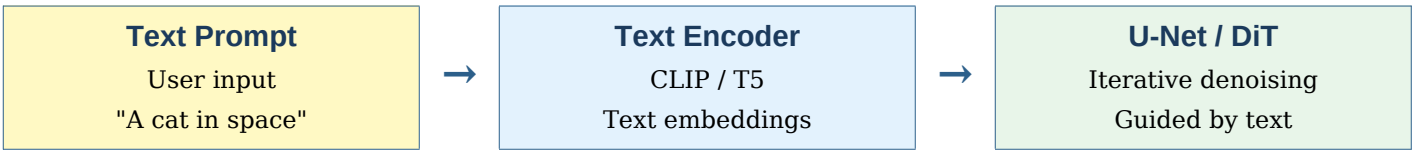


*Figure: Simplified Stable Diffusion Pipeline Architecture*

```
from diffusers import StableDiffusionPipeline
import torch

# Load Stable Diffusion pipeline
pipe = StableDiffusionPipeline.from_pretrained(
    'runwayml/stable-diffusion-v1-5',
    torch_dtype=torch.float16,
```

```
)
pipe = pipe.to('cuda')

# Generate an image
prompt = 'A serene mountain lake at sunset, photorealistic'
image = pipe(prompt, num_inference_steps=50).images[0]
image.save('generated_image.png')
```

## 3.7 The Hugging Face Hub

The Hugging Face Hub is the central repository that ties the entire ecosystem together. Think of it as a combination of GitHub, Docker Hub, and an app store, all tailored for machine learning. Every item on the Hub has a dedicated page with documentation, usage statistics, community discussions, and interactive widgets for trying models directly in the browser.

| Hub Feature | Description | Research Benefit |
|---|---|---|
| Model Cards | Standardized documentation for models | Understanding capabilities and limitations |
| Dataset Cards | Documentation for datasets | Data provenance and licensing |
| Git Versioning | Full version control for all artifacts | Reproducibility and tracking |
| Discussions | Community Q&A on each artifact | Collaborative problem-solving |
| Spaces | Interactive demo hosting | Showcasing research |
| Organizations | Team collaboration features | Research group management |
| Inference Widget | Try models in the browser | Quick evaluation without code |
| Leaderboards | Benchmark rankings | Model comparison |

*Figure: Hugging Face Hub Features for Researchers*

### 🔬 LAB: Exploring the Hugging Face Ecosystem in Action

In this lab, you will use multiple Hugging Face libraries together to build a simple end-to-end workflow: load a dataset, tokenize it, run inference with a pre-trained model, and evaluate the results.

```
# Lab 3: End-to-End Ecosystem Walkthrough
```

```
from datasets import load_dataset
from transformers import pipeline
import evaluate

# Step 1: Load a subset of the IMDB dataset
dataset = load_dataset('imdb',
split='test').shuffle(seed=42).select(range(100))
print(f'Loaded {len(dataset)} examples')

# Step 2: Create a sentiment analysis pipeline
classifier = pipeline(
    'sentiment-analysis',
    model='distilbert-base-uncased-finetuned-sst-2-english',
    device=0
)

# Step 3: Run inference on all examples
predictions = classifier(dataset['text'], batch_size=32, truncation=True)

# Step 4: Convert predictions to numeric labels
pred_labels = [1 if p['label'] == 'POSITIVE' else 0 for p in predictions]
true_labels = dataset['label']

# Step 5: Evaluate
metrics = evaluate.combine(['accuracy', 'f1', 'precision', 'recall'])
results = metrics.compute(predictions=pred_labels, references=true_labels)

print('\n=== Evaluation Results ===')
for metric, value in results.items():
    print(f'{metric:>12}: {value:.4f}')
```

**Expected Output:** You should see accuracy above 0.85 and F1 score above 0.84. The exact numbers may vary slightly due to the random sample, but the model should perform well on this well-known benchmark dataset.

| Metric | Expected Range |
|---|---|
| Accuracy | 0.85 – 0.92 |
| F1 | 0.84 – 0.92 |
| Precision | 0.83 – 0.93 |

| Metric | Expected Range |
|---|---|
| Recall | 0.82 – 0.92 |

*Figure: Expected metric ranges for Lab 3*

## Deep Dive: Pipeline Internals

Understanding how pipelines work internally helps you use them more effectively and debug issues when they arise. A pipeline performs three main operations in sequence: preprocessing, forward pass, and postprocessing. Let us examine each in detail.

During preprocessing, the pipeline converts your raw input into the format expected by the model. For text inputs, this means tokenization with appropriate padding and truncation. For image inputs, it means resizing, normalization, and conversion to tensors. For audio inputs, it means resampling to the correct sample rate and computing spectrograms or waveform features.

The forward pass sends the preprocessed input through the model and returns raw outputs. For classification models, these are logits. For generation models, these are token probabilities at each step. For detection models, these are bounding box coordinates and class predictions.

Postprocessing converts the raw model outputs into human-readable results. This includes applying softmax to get probabilities, decoding token IDs back to text, filtering low-confidence detections, and formatting the output as a list of dictionaries with standardized keys.

```
# Looking inside a pipeline
from transformers import pipeline

# Create pipeline with verbose output
pipe = pipeline('sentiment-analysis', device=0)

# Access internal components
print(f'Model: {type(pipe.model).__name__}')
print(f'Tokenizer: {type(pipe.tokenizer).__name__}')
print(f'Device: {pipe.device}')
print(f'Framework: {pipe.framework}')

# Manual pipeline steps
text = 'I really enjoy learning about transformers!'

# Step 1: Preprocess (tokenize)
inputs = pipe.tokenizer(text, return_tensors='pt', truncation=True)
inputs = {k: v.to(pipe.device) for k, v in inputs.items()}
print(f'\nPreprocessed: {inputs["input_ids"].shape}')

# Step 2: Forward pass
```

```
import torch
with torch.no_grad():
    outputs = pipe.model(**inputs)
print(f'Raw logits: {outputs.logits}')

# Step 3: Postprocess
probs = torch.softmax(outputs.logits, dim=-1)
pred_idx = probs.argmax().item()
label = pipe.model.config.id2label[pred_idx]
score = probs[0][pred_idx].item()
print(f'Result: {label} ({score:.4f})')

# Compare with pipeline output
print(f'\nPipeline: {pipe(text)}')
```

## Framework Comparison: PyTorch vs TensorFlow vs JAX

Hugging Face supports three deep learning frameworks. While all three work with the Transformers library, each has different strengths that may influence your choice. In Colab, PyTorch is the default and most well-supported option.

| Feature | PyTorch | TensorFlow | JAX |
|---|---|---|---|
| Colab Support | Excellent (default) | Good | Experimental |
| HF Model Availability | ~95% of models | ~60% of models | ~20% of models |
| Debugging | Eager mode (easy) | Graph mode (harder) | Functional (different) |
| Community Size | Largest in research | Large in production | Growing |
| Dynamic Graphs | Yes (native) | Yes (tf.function) | Yes (jit) |
| Memory Efficiency | Good | Good | Excellent |
| Training Speed | Fast | Fast | Very fast (TPU) |
| Recommended For | Research, prototyping | Production, serving | TPU workloads |

*Figure: Deep Learning Framework Comparison for Hugging Face*

### Understanding Model Cards in Detail

Model cards are the research papers of the Hugging Face Hub. A well-written model card contains all the information needed to understand, evaluate, and responsibly use a model. Learning to read model cards efficiently is a critical skill for any AI researcher.

| Section | What to Look For | Why It Matters |
|---|---|---|
| Model Description | Architecture, size, base model | Understand what you are using |
| Intended Use | Recommended tasks, languages | Ensure task compatibility |
| Training Data | Datasets, preprocessing, size | Assess domain relevance and bias |
| Training Procedure | Hyperparameters, hardware, time | Reproducibility |
| Evaluation Results | Metrics, benchmarks, comparisons | Performance expectations |
| Limitations | Known failure modes, biases | Avoid misuse |
| Ethical Considerations | Risks, mitigation strategies | Responsible deployment |
| Carbon Footprint | Training emissions estimate | Environmental awareness |

*Figure: Model Card Anatomy: Sections and What to Look For*

## Part I Capstone Lab

### 🔬 LAB: Building Your Research Starter Kit

This comprehensive lab combines everything from Part I into a single, reusable research notebook. By the end, you will have a template that you can copy for every new research project.

**Objective:** Create a fully configured research environment notebook with GPU verification, package installation, Drive mounting, Hugging Face authentication, utility functions, and a quick sanity test. Save this as your permanent template.

**Duration:** 45–60 minutes

**Prerequisites:** Google account, Hugging Face account with access token

### Task 1: Environment Configuration (15 min)

```
# ============================================
# RESEARCH STARTER KIT - Template Notebook
# ============================================
# Project: [YOUR PROJECT NAME]
# Date:    [TODAY'S DATE]
# Author:  [YOUR NAME]
# Goal:    [RESEARCH QUESTION]
# ============================================

# --- Install Packages ---
!pip install -q transformers==4.40.0 datasets==2.19.0
!pip install -q evaluate==0.4.1 accelerate==0.30.0
!pip install -q peft==0.10.0 bitsandbytes==0.43.0
!pip install -q sentence-transformers==2.7.0 faiss-cpu
!pip install -q gradio==4.25.0 huggingface_hub

import sys
print(f'Python: {sys.version}')
print('All packages installed successfully!')
```

### Task 2: GPU and Resource Verification (10 min)

```
import torch, psutil, shutil

def full_resource_check():
    print('=' * 50)
    print('RESOURCE CHECK')
    print('=' * 50)

    # CPU
    print(f'\nCPU Cores:     {psutil.cpu_count()}')
    ram = psutil.virtual_memory()
    print(f'RAM:           {ram.total/1e9:.1f} GB total,
{ram.available/1e9:.1f} GB free')

    # Disk
```

```
    disk = shutil.disk_usage('/')
    print(f'Disk:          {disk.total/1e9:.0f} GB total,
{disk.free/1e9:.0f} GB free')

    # GPU
    if torch.cuda.is_available():
        gpu = torch.cuda.get_device_properties(0)
        print(f'\nGPU:           {gpu.name}')
        print(f'VRAM:          {gpu.total_mem/1e9:.1f} GB')
        print(f'CUDA Version:  {torch.version.cuda}')
        print(f'PyTorch:       {torch.__version__}')
        print(f'\nStatus: READY FOR DEEP LEARNING')
    else:
        print('\nWARNING: No GPU detected!')
        print('Go to Runtime > Change runtime type > GPU')

full_resource_check()
```

## Task 3: Drive and Authentication Setup (10 min)

```
# Mount Google Drive
from google.colab import drive
drive.mount('/content/drive')

# Create project directory structure
import os
PROJECT = '/content/drive/MyDrive/hf_research'
DIRS = ['data', 'models', 'results', 'notebooks', 'cache', 'logs']
for d in DIRS:
    path = f'{PROJECT}/{d}'
    os.makedirs(path, exist_ok=True)
print(f'Project root: {PROJECT}')
print(f'Subdirectories: {DIRS}')

# Set HF cache to persistent storage
os.environ['HF_HOME'] = f'{PROJECT}/cache'

# Authenticate with Hugging Face
from huggingface_hub import login
login()  # Enter your token
```

### Task 4: Utility Functions Library (10 min)

```
# Reusable utility functions
import time, json, torch
from datetime import datetime

def timer(func):
    '''Decorator to time function execution'''
    def wrapper(*args, **kwargs):
        start = time.time()
        result = func(*args, **kwargs)
        elapsed = time.time() - start
        print(f'{func.__name__}: {elapsed:.2f}s')
        return result
    return wrapper

def gpu_mem():
    '''Quick GPU memory report'''
    if torch.cuda.is_available():
        a = torch.cuda.memory_allocated()/1e9
        r = torch.cuda.memory_reserved()/1e9
        t = torch.cuda.get_device_properties(0).total_mem/1e9
        print(f'GPU: {a:.2f}/{t:.1f} GB allocated ({a/t*100:.0f}%)')

def save_results(results, name, base=PROJECT):
    '''Save experiment results to Drive'''
    path = f'{base}/results/{name}_{datetime.now():%Y%m%d_%H%M}.json'
    with open(path, 'w') as f:
        json.dump(results, f, indent=2, default=str)
    print(f'Results saved: {path}')
    return path

def count_params(model):
    '''Count total and trainable parameters'''
    total = sum(p.numel() for p in model.parameters())
    trainable = sum(p.numel() for p in model.parameters() if
p.requires_grad)
    print(f'Total params:     {total:>12,}')
    print(f'Trainable params: {trainable:>12,} ({trainable/total*100:.1f}
%)')

print('Utility functions loaded!')
```

## Task 5: Sanity Test — End-to-End Inference (10 min)

```
from transformers import pipeline

@timer
def run_sanity_test():
    # Test 1: Sentiment Analysis
    clf = pipeline('sentiment-analysis', device=0)
    res = clf('This research environment is working perfectly!')
    assert res[0]['label'] in ['POSITIVE', 'NEGATIVE']
    print(f'Test 1 PASSED: Sentiment = {res[0]["label"]} ({res[0]
["score"]:.4f})')

    # Test 2: Dataset Loading
    from datasets import load_dataset
    ds = load_dataset('imdb', split='train[:10]')
    assert len(ds) == 10
    print(f'Test 2 PASSED: Loaded {len(ds)} IMDB examples')

    # Test 3: Evaluation
    import evaluate
    acc = evaluate.load('accuracy')
    r = acc.compute(predictions=[1,0,1], references=[1,0,1])
    assert r['accuracy'] == 1.0
    print(f'Test 3 PASSED: Accuracy metric works')

    # Test 4: Drive Writing
    save_results({'test': True, 'status': 'all_passed'}, 'sanity_test')
    print(f'Test 4 PASSED: Drive writing works')

    gpu_mem()
    print('\nALL SANITY TESTS PASSED!')
    print('Your research environment is fully configured.')

run_sanity_test()
```

| Checklist Item | Status |
|---|---|
| Packages installed with pinned versions | ☐ |
| GPU detected and verified | ☐ |

| Checklist Item | Status |
|---|---|
| Google Drive mounted and project structure created | ☐ |
| HF_HOME set to Drive cache | ☐ |
| Hugging Face authentication successful | ☐ |
| Utility functions loaded | ☐ |
| All 4 sanity tests passed | ☐ |
| Template notebook saved to Drive/notebooks/ | ☐ |

*Figure: Part I Capstone Lab Completion Checklist*

# PART II

## WORKING WITH MODELS

# Chapter 4. Running Your First Model in Google Colab

This chapter is where theory meets practice. You will load your first pre-trained model from the Hugging Face Hub, tokenize input text, run inference, and interpret the results. By the end of this chapter, you will be comfortable running any model from the Hub and will understand the fundamental workflow that underpins all subsequent chapters.

## 4.1 Loading a Model

Loading a pre-trained model from the Hugging Face Hub is the first step in any research workflow. The process involves three components: downloading the model weights, loading the configuration, and initializing the tokenizer. The Auto classes handle all of this automatically.

When you call from_pretrained with a model name, the library downloads the model weights and configuration from the Hub, caches them locally, and initializes the model object. On Colab, the cache is stored in the runtime's temporary storage at ~/.cache/huggingface/. If you want to persist the cache across sessions, point HF_HOME to your mounted Google Drive.

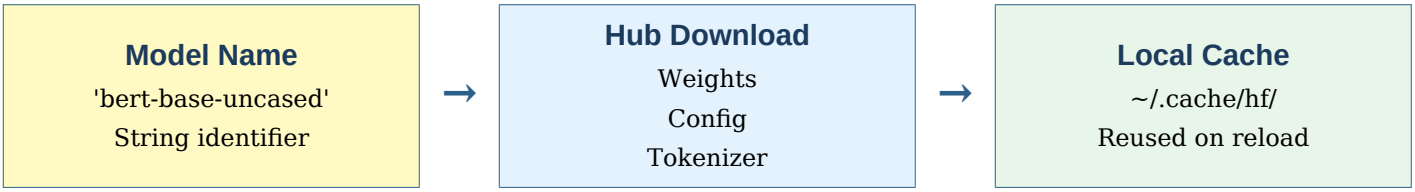


*Figure: Model loading workflow: from Hub identifier to cached local model*

```
from transformers import AutoModelForSequenceClassification, AutoTokenizer
import torch

# Load model and tokenizer
model_name = 'distilbert-base-uncased-finetuned-sst-2-english'
tokenizer = AutoTokenizer.from_pretrained(model_name)
model = AutoModelForSequenceClassification.from_pretrained(model_name)

# Move model to GPU
device = 'cuda' if torch.cuda.is_available() else 'cpu'
model = model.to(device)
model.eval()  # Set to evaluation mode
```

```
# Print model info
total_params = sum(p.numel() for p in model.parameters())
print(f'Model: {model_name}')
print(f'Parameters: {total_params:,}')
print(f'Device: {device}')
```

| Model Size | Parameters | Memory (FP32) | Memory (FP16) | Colab Compatible? |
|---|---|---|---|---|
| Tiny | < 50M | < 200 MB | < 100 MB | Yes (Free tier) |
| Small | 50M – 200M | 200 MB – 800 MB | 100 – 400 MB | Yes (Free tier) |
| Base | 200M – 500M | 800 MB – 2 GB | 400 MB – 1 GB | Yes (Free tier) |
| Large | 500M – 3B | 2 – 12 GB | 1 – 6 GB | Yes (may need quantization) |
| XL | 3B – 13B | 12 – 52 GB | 6 – 26 GB | Requires quantization |
| XXL | 13B+ | 52+ GB | 26+ GB | 4-bit quantization required |

*Figure: Model Size Guide — Memory requirements and Colab compatibility*

## 4.2 Tokenization

Before text can be fed to a model, it must be tokenized. Tokenization converts raw text strings into sequences of numerical IDs that correspond to entries in the model's vocabulary. Each model has its own tokenizer that was trained alongside it, so it is critical to always use the matching tokenizer.

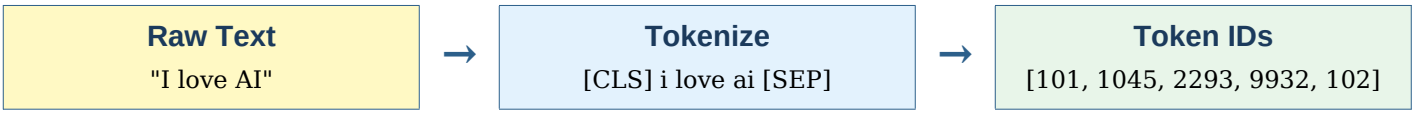


*Figure: Tokenization process: from raw text to numerical IDs*

```
# Tokenization in detail
text = 'Hugging Face is revolutionizing AI research!'

# Step 1: Tokenize to subwords
tokens = tokenizer.tokenize(text)
```

```
print(f'Tokens: {tokens}')
# ['hugging', 'face', 'is', 'revolution', '##izing', 'ai', 'research',
'!']

# Step 2: Convert to IDs with special tokens
encoded = tokenizer(text, return_tensors='pt')
print(f'Input IDs shape: {encoded["input_ids"].shape}')
print(f'Input IDs: {encoded["input_ids"]}')
print(f'Attention mask: {encoded["attention_mask"]}')

# Step 3: Decode back to text
decoded = tokenizer.decode(encoded['input_ids'][0])
print(f'Decoded: {decoded}')
```

| Tokenizer Output | Type | Shape | Description |
|---|---|---|---|
| input_ids | tensor | [batch, seq_len] | Numerical token IDs |
| attention_mask | tensor | [batch, seq_len] | 1 for real tokens, 0 for padding |
| token_type_ids | tensor | [batch, seq_len] | Segment IDs (BERT-style models only) |
| offset_mapping | list | [batch, seq_len, 2] | Character offsets (optional) |
| special_tokens_mask | list | [batch, seq_len] | 1 for special tokens (optional) |

*Figure: Tokenizer Output Components*

## 4.3 Inference Basics

Running inference means passing tokenized input through the model to obtain predictions. For classification models, the output contains logits, which are raw unnormalized scores for each possible class. To convert logits to probabilities, apply the softmax function. For generation models, the output process is more complex and involves autoregressive decoding.

```
# Complete inference workflow
import torch
import torch.nn.functional as F

# Prepare input
text = 'This movie was absolutely fantastic!'
inputs = tokenizer(text, return_tensors='pt').to(device)
```

```
# Run inference (disable gradient computation for efficiency)
with torch.no_grad():
    outputs = model(**inputs)

# Process outputs
logits = outputs.logits
probabilities = F.softmax(logits, dim=-1)
predicted_class = torch.argmax(probabilities, dim=-1).item()

# Map to labels
labels = model.config.id2label
print(f'Text: {text}')
print(f'Prediction: {labels[predicted_class]}')
print(f'Confidence: {probabilities[0][predicted_class]:.4f}')
print(f'All probabilities: {dict(zip(labels.values(),
probabilities[0].tolist()))}')
```

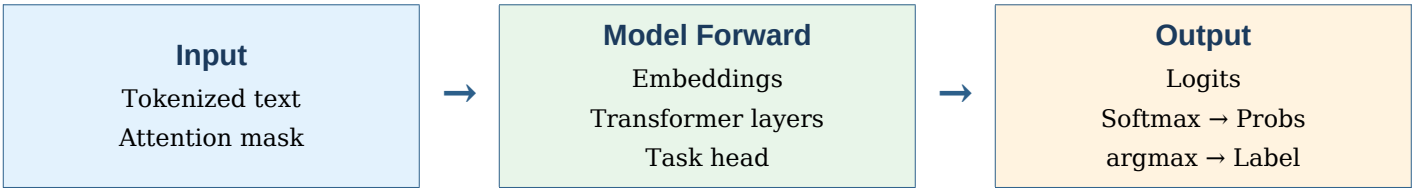


*Figure: Inference pipeline: from tokenized input to predicted label*

## 4.4 The Pipeline API

While the manual approach gives you full control, the Pipeline API provides a much more convenient interface for common tasks. A pipeline encapsulates the tokenizer, model, and post-processing logic into a single callable object. It is the fastest way to go from raw input to useful output.

| Task Name | Pipeline String | Input | Output |
|---|---|---|---|
| Sentiment Analysis | 'sentiment-analysis' | Text | Label + score |
| NER | 'ner' | Text | Entities + positions |
| Question Answering | 'question-answering' | Question + context | Answer + score |
| Summarization | 'summarization' | Long text | Short summary |

| Task Name | Pipeline String | Input | Output |
|---|---|---|---|
| Translation | 'translation_en_to_fr' | Source text | Translated text |
| Text Generation | 'text-generation' | Prompt | Generated text |
| Fill Mask | 'fill-mask' | Text with [MASK] | Predictions |
| Zero-Shot Classification | 'zero-shot-classification' | Text + labels | Label + scores |
| Image Classification | 'image-classification' | Image | Label + score |
| Object Detection | 'object-detection' | Image | Objects + boxes |
| Speech Recognition | 'automatic-speech-recognition' | Audio | Transcription |
| Text-to-Speech | 'text-to-speech' | Text | Audio waveform |

*Figure: Available Pipeline Tasks in the Transformers Library*

```
from transformers import pipeline

# Quick setup with default model
classifier = pipeline('sentiment-analysis', device=0)

# Single prediction
result = classifier('This book is incredibly helpful!')
print(result)  # [{'label': 'POSITIVE', 'score': 0.9998}]

# Batch prediction (much faster)
texts = [
    'I love this product, it works perfectly!',
    'Terrible quality, broke after one day.',
    'It is okay, nothing special but functional.',
    'Best purchase I have ever made!',
    'Complete waste of money, avoid at all costs.',
]
results = classifier(texts, batch_size=8)
for text, res in zip(texts, results):
    print(f'{res["label"]:>8} ({res["score"]:.4f}): {text[:50]}')
```

> ✓ **Tip:** *Always set device=0 (or device='cuda:0') when creating a pipeline in Colab to ensure GPU acceleration. Without this, the pipeline runs on CPU, which is 10-100x slower for transformer models.*

## 🔬 LAB: Multi-Task Inference Pipeline

In this lab, you will create multiple pipelines for different tasks and run them on various inputs. This will demonstrate the versatility of the Pipeline API and help you understand which task types are available.

```
# Lab 4: Multi-Task Inference
from transformers import pipeline

# Task 1: Sentiment Analysis
print('=== SENTIMENT ANALYSIS ===')
sentiment = pipeline('sentiment-analysis', device=0)
texts = ['I am so happy today!', 'This is the worst day ever.', 'The
weather is mild.']
for text in texts:
    res = sentiment(text)[0]
    print(f'  {res["label"]:>8} ({res["score"]:.3f}): {text}')

# Task 2: Named Entity Recognition
print('\n=== NAMED ENTITY RECOGNITION ===')
ner = pipeline('ner', aggregation_strategy='simple', device=0)
text = 'Barack Obama was born in Hawaii and served as the 44th President.'
entities = ner(text)
for ent in entities:
    print(f'  {ent["entity_group"]:>6}: {ent["word"]}
({ent["score"]:.3f})')

# Task 3: Question Answering
print('\n=== QUESTION ANSWERING ===')
qa = pipeline('question-answering', device=0)
context = 'The Transformers library was created by Hugging Face in 2018.'
questions = ['Who created Transformers?', 'When was it created?']
for q in questions:
    ans = qa(question=q, context=context)
    print(f'  Q: {q}')
    print(f'  A: {ans["answer"]} ({ans["score"]:.3f})')

# Task 4: Text Generation
print('\n=== TEXT GENERATION ===')
```

```
generator = pipeline('text-generation', model='gpt2', device=0)
prompt = 'The future of artificial intelligence is'
output = generator(prompt, max_length=60, do_sample=True, temperature=0.7)
print(f'  Prompt: {prompt}')
print(f'  Output: {output[0]["generated_text"]}')

# Task 5: Zero-Shot Classification
print('\n=== ZERO-SHOT CLASSIFICATION ===')
zsc = pipeline('zero-shot-classification', device=0)
text = 'The new quantum processor achieves record-breaking performance.'
labels = ['technology', 'sports', 'politics', 'science', 'entertainment']
result = zsc(text, candidate_labels=labels)
for label, score in zip(result['labels'], result['scores']):
    bar = '#' * int(score * 30)
    print(f'  {label:>15}: {bar} ({score:.3f})')
```

Record the outputs of each task in the table below and analyze the model's performance:

| Task | Input Summary | Output Quality (1-5) | Notes |
|---|---|---|---|
| Sentiment Analysis | 3 texts with clear sentiment | (Rate) | (Your observations) |
| NER | Sentence with named entities | (Rate) | (Your observations) |
| Question Answering | 2 factual questions | (Rate) | (Your observations) |
| Text Generation | Open-ended prompt | (Rate) | (Your observations) |
| Zero-Shot Classification | Tech news sentence | (Rate) | (Your observations) |

*Figure: Lab 4 Evaluation Sheet*

## Understanding Model Architectures

Before diving deeper into using models, it is valuable to understand the main architectural families you will encounter on the Hugging Face Hub. Each architecture has different strengths and is suited to different tasks. Knowing which architecture to choose for your task can save hours of experimentation.

The encoder-only architecture, exemplified by BERT, processes the entire input at once and produces contextualized representations. Each token's representation is informed by all other tokens in the input. This bidirectional context makes encoder models excellent for understanding tasks like classification, named entity recognition, and extractive question answering.

The decoder-only architecture, exemplified by GPT, processes tokens from left to right and generates one token at a time. Each token can only attend to previous tokens, not future ones. This autoregressive nature makes decoder models ideal for text generation, code completion, and conversational AI.

The encoder-decoder architecture, exemplified by T5 and BART, uses an encoder to process the input and a decoder to generate the output. This two-stage approach is well-suited for tasks where the output is a transformation of the input, such as translation, summarization, and text-to-text conversion.

| Architecture | Key Models | Direction | Best Tasks | Input/Output |
|---|---|---|---|---|
| Encoder-only | BERT, RoBERTa, DistilBERT, ELECTRA, DeBERTa | Bidirectional | Classification, NER, QA | Text → Embeddings/Labels |
| Decoder-only | GPT-2, LLaMA, Mistral, Falcon, Phi | Left-to-right | Generation, completion | Prompt → Text |
| Encoder-Decoder | T5, BART, mBART, Pegasus, FLAN-T5 | Bi + Left-to-right | Translation, summarization | Text → Text |
| Vision Transformer | ViT, DeiT, BEiT, Swin, ConvNeXt | Bidirectional | Image tasks | Image → Labels/Features |
| Multimodal | CLIP, BLIP, LLaVA, Flamingo | Varies | Vision + language | Image+Text → Text |

*Figure: Model Architecture Families and Their Characteristics*

## Model Size vs Performance: The Scaling Law Perspective

One of the most important discoveries in modern AI research is the existence of scaling laws. These empirical relationships show that model performance improves predictably as model size, dataset size, and compute budget increase. Understanding scaling laws helps you make informed decisions about which model size to use for your task.

For practical research in Colab, the key insight from scaling laws is that larger models are not always better for every task. On many benchmarks, a well-fine-tuned small model can outperform a much larger model running zero-shot or few-shot. The sweet spot depends on how much labeled data you have, how much compute you can afford, and how much accuracy you need.

| Model Scale | Example Models | Typical Use in Colab | Fine-Tuning Approach |
|---|---|---|---|
| Tiny (< 100M) | DistilBERT (66M), TinyBERT (14M) | Full fine-tuning, fast iteration | Standard Trainer, no special tricks |
| Small (100-500M) | BERT-base (110M), RoBERTa (125M) | Full or LoRA fine-tuning | FP16, moderate batch size |
| Medium (500M-3B) | BERT-large (340M), Flan-T5-XL (3B) | LoRA fine-tuning | FP16/BF16, gradient accumulation |
| Large (3-13B) | LLaMA-7B, Mistral-7B | QLoRA (4-bit + LoRA) | 4-bit quantization required |
| XL (13B+) | LLaMA-13B, Mixtral-8x7B | Inference only (quantized) | Cannot fine-tune in Colab |

*Figure: Model Scale Guide for Google Colab*

## Practical Model Selection Flowchart

When starting a new project, use the following decision process to select the right model:

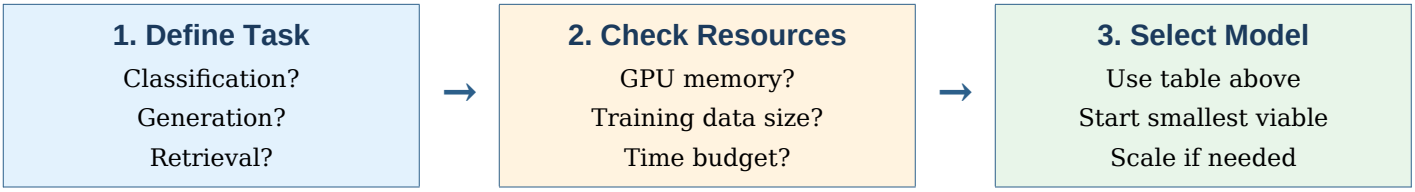


*Figure: Model selection decision process*

## 🔬 LAB: Model Architecture Comparison Lab

In this extended lab, you will load three different model architectures and compare their behavior on the same input. This hands-on comparison will solidify your understanding of the differences between encoder, decoder, and encoder-decoder models.

```
# Lab: Comparing Model Architectures
from transformers import AutoModel, AutoModelForSequenceClassification
from transformers import AutoModelForCausalLM, AutoModelForSeq2SeqLM
from transformers import AutoTokenizer
import torch

text = 'Artificial intelligence is transforming scientific research.'

# === Encoder Model (BERT) ===
print('=== ENCODER MODEL (BERT) ===')
enc_name = 'bert-base-uncased'
enc_tok = AutoTokenizer.from_pretrained(enc_name)
enc_model = AutoModel.from_pretrained(enc_name)
enc_inputs = enc_tok(text, return_tensors='pt')
with torch.no_grad():
    enc_out = enc_model(**enc_inputs)
print(f'Model: {enc_name}')
print(f'Parameters: {sum(p.numel() for p in enc_model.parameters()):,}')
print(f'Input tokens: {enc_inputs["input_ids"].shape[1]}')
print(f'Hidden state shape: {enc_out.last_hidden_state.shape}')
print(f'  -> Each token gets a {enc_out.last_hidden_state.shape[-1]}-dim
representation')
print(f'  -> All tokens see all other tokens (bidirectional)')

# === Decoder Model (GPT-2) ===
print('\n=== DECODER MODEL (GPT-2) ===')
dec_name = 'gpt2'
dec_tok = AutoTokenizer.from_pretrained(dec_name)
dec_model = AutoModelForCausalLM.from_pretrained(dec_name)
dec_inputs = dec_tok(text, return_tensors='pt')
with torch.no_grad():
    dec_out = dec_model(**dec_inputs)
print(f'Model: {dec_name}')
print(f'Parameters: {sum(p.numel() for p in dec_model.parameters()):,}')
print(f'Logits shape: {dec_out.logits.shape}')
print(f'  -> At each position, predicts next token from
{dec_out.logits.shape[-1]} vocabulary')
print(f'  -> Each token only sees previous tokens (left-to-right)')

# Generate continuation
from transformers import pipeline
```

```
gen = pipeline('text-generation', model=dec_name)
continuation = gen(text, max_new_tokens=30, do_sample=False)[0]
['generated_text']
print(f'\nGenerated: {continuation}')

# === Encoder-Decoder Model (T5) ===
print('\n=== ENCODER-DECODER MODEL (T5) ===')
ed_name = 't5-small'
ed_tok = AutoTokenizer.from_pretrained(ed_name)
ed_model = AutoModelForSeq2SeqLM.from_pretrained(ed_name)
print(f'Model: {ed_name}')
print(f'Parameters: {sum(p.numel() for p in ed_model.parameters()):,}')

# Summarization with T5
input_text = f'summarize: {text}'
ed_inputs = ed_tok(input_text, return_tensors='pt')
with torch.no_grad():
    ed_out = ed_model.generate(**ed_inputs, max_new_tokens=50)
summary = ed_tok.decode(ed_out[0], skip_special_tokens=True)
print(f'Input: {input_text}')
print(f'Output: {summary}')
print(f'  -> Encoder reads input, decoder generates output token by
token')
```

| Property | BERT (Encoder) | GPT-2 (Decoder) | T5 (Enc-Dec) |
|---|---|---|---|
| Parameters | 110M | 124M | 60M |
| Attention | Bidirectional | Causal (left-to-right) | Bi (enc) + Causal (dec) |
| Output type | Embeddings per token | Next-token probabilities | Generated sequence |
| Main strength | Understanding | Generation | Transformation |
| Memory per token | Lower (no generation) | Medium | Higher (two components) |

*Figure: Side-by-side comparison of three architecture families*

# Chapter 5. Natural Language Processing with Transformers

Natural Language Processing is the most mature and widely used application of transformer models. This chapter covers the five core NLP tasks in detail: text classification, summarization, translation, question answering, and text generation. For each task, you will learn the theoretical background, see practical code examples, understand common model choices, and work through guided exercises.

## 5.1 Text Classification

Text classification assigns one or more labels to a piece of text. It is the most common NLP task and encompasses a wide range of applications including sentiment analysis, topic categorization, spam detection, intent recognition, and toxicity detection.

The typical approach for text classification with transformers involves taking a pre-trained language model and adding a classification head on top. The classification head is a simple linear layer that maps the model's hidden representation to the number of classes. During fine-tuning, both the pre-trained weights and the classification head are updated.

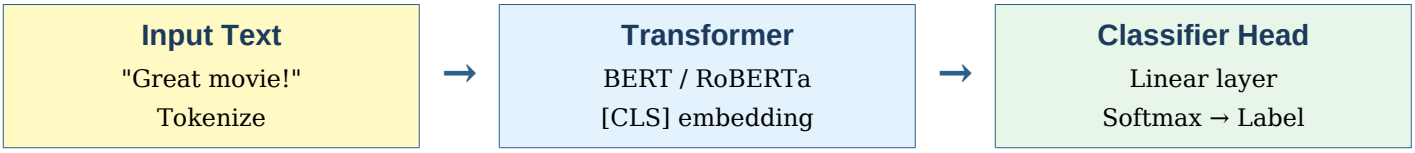


*Figure: Text classification architecture with transformer encoder*

| Classification Type | Description | Example | Common Models |
|---|---|---|---|
| Binary | Two classes | Positive/Negative sentiment | DistilBERT, BERT |
| Multi-class | Multiple exclusive classes | News topic (Sports/Tech/Politics) | RoBERTa, XLNet |
| Multi-label | Multiple non-exclusive | Article tags (AI, Ethics, Research) | BERT + sigmoid |
| Zero-shot | No task-specific training | Classify with any labels | BART-MNLI, DeBERTa |

*Figure: Types of Text Classification Tasks*

```
# Advanced classification with confidence analysis
from transformers import pipeline
import torch

# Multi-class classification with top-k results
classifier = pipeline(
    'text-classification',
    model='distilbert-base-uncased-finetuned-sst-2-english',
    device=0,
    top_k=None  # Return all class scores
)

texts = [
    'This is the best movie I have ever seen!',
    'The acting was mediocre but the plot was interesting.',
    'Absolutely terrible. A complete waste of two hours.',
    'I am not sure how I feel about this film.',
]

for text in texts:
    results = classifier(text)
    print(f'\nText: {text[:60]}')
    for r in results[0]:
        bar = '|' * int(r['score'] * 40)
        print(f'  {r["label"]:>10}: {bar} {r["score"]:.4f}')
```

## 5.2 Summarization

Automatic text summarization condenses long documents into shorter versions while preserving key information. This is one of the most practically useful NLP tasks, especially for researchers who need to process large volumes of literature.

Modern transformer models perform abstractive summarization, meaning they generate new text rather than simply extracting sentences from the original. Models like BART, T5, and Pegasus have achieved impressive performance on standard benchmarks like CNN/DailyMail and XSum.

| Model | Architecture | Max Input | Training Data | Strengths |
|---|---|---|---|---|
| BART-large-CNN | Encoder-Decoder | 1024 tokens | CNN/DailyMail | News articles, balanced |
| T5-base | Encoder-Decoder | 512 tokens | C4 | Multi-task, flexible |
| Pegasus-large | Encoder-Decoder | 1024 tokens | Multiple news | News summarization |
| LED-base | Longformer E-D | 16384 tokens | arXiv + others | Long documents |
| BART-large-XSum | Encoder-Decoder | 1024 tokens | XSum | Extreme summarization |

*Figure: Popular Summarization Models on Hugging Face Hub*

```
from transformers import pipeline

summarizer = pipeline('summarization', model='facebook/bart-large-cnn',
device=0)

article = '''
Artificial intelligence has transformed nearly every industry in the past
decade.
From healthcare to finance, from education to entertainment, AI systems
are being
deployed at an unprecedented scale. The rise of large language models in
particular
has captured the public imagination, with systems like GPT and Claude
demonstrating
remarkable capabilities in understanding and generating human language.
Researchers
are now exploring the boundaries of what these models can do, from
scientific
discovery to creative writing. However, significant challenges remain,
including
concerns about bias, safety, energy consumption, and the economic impact
of
widespread automation. The next decade will likely see even more dramatic
advances
as computing power continues to grow and new architectural innovations
emerge.
'''
```

```
# Generate summary with different length settings
for max_len in [50, 100, 150]:
    summary = summarizer(article, max_length=max_len, min_length=20)
    print(f'\n--- Max {max_len} tokens ---')
    print(summary[0]['summary_text'])
```

## 5.3 Translation

Machine translation has achieved remarkable quality with transformer models. The Helsinki-NLP collection on Hugging Face provides translation models for hundreds of language pairs, and multilingual models like mBART and M2M100 can translate between many language pairs with a single model.

```
from transformers import pipeline

# English to French translation
en_fr = pipeline('translation', model='Helsinki-NLP/opus-mt-en-fr',
device=0)
result = en_fr('Machine learning is transforming scientific research.')
print(f'EN: Machine learning is transforming scientific research.')
print(f'FR: {result[0]["translation_text"]}')

# English to German
en_de = pipeline('translation', model='Helsinki-NLP/opus-mt-en-de',
device=0)
result = en_de('The experiment produced unexpected results.')
print(f'DE: {result[0]["translation_text"]}')
```

## 5.4 Question Answering

Extractive question answering identifies the span of text within a context that answers a given question. This is different from generative QA where the model creates an answer from scratch. Extractive QA is well-suited for situations where the answer is explicitly stated in a document.

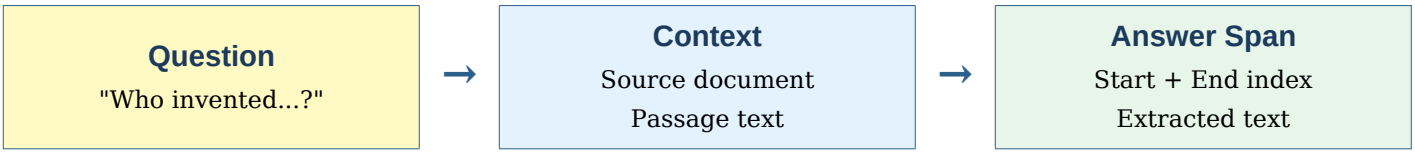

*Figure: Extractive Question Answering Architecture*

```
qa = pipeline('question-answering', device=0)

context = '''
The Transformer architecture was introduced in the paper "Attention Is All You Need"
by Vaswani et al. in 2017. It replaced recurrent neural networks with self-attention
mechanisms, enabling much faster training through parallelization. The architecture
consists of an encoder and decoder, each made up of multiple layers of multi-head
attention and feed-forward networks. BERT, which uses only the encoder part, was
released by Google in 2018 and quickly became the dominant model for NLP tasks.
'''

questions = [
    'When was the Transformer introduced?',
    'What did the Transformer replace?',
    'Who created the Transformer?',
    'When was BERT released?',
    'What parts does the architecture consist of?',
]

for q in questions:
    ans = qa(question=q, context=context)
    print(f'Q: {q}')
    print(f'A: {ans["answer"]} (confidence: {ans["score"]:.4f})')
    print()
```

## 5.5 Text Generation

Text generation is one of the most exciting applications of transformer models. Autoregressive models generate text one token at a time, with each new token conditioned on all previously generated tokens. The quality of generated text depends on both the model's training and the decoding strategy used.

| Decoding Strategy | Description | Temperature | Best For |
|---|---|---|---|
| Greedy | Always pick highest probability token | N/A | Deterministic, factual |
| Beam Search | Track top-k sequences | N/A | Translation, summarization |
| Top-k Sampling | Sample from top k tokens | 0.5–1.5 | Creative, diverse text |
| Top-p (Nucleus) | Sample from cumulative probability p | 0.5–1.5 | Natural, varied text |
| Contrastive Search | Balance quality and diversity | N/A | Coherent long text |

*Figure: Text Generation Decoding Strategies*

```
from transformers import pipeline

generator = pipeline('text-generation', model='gpt2', device=0)

prompt = 'In the year 2050, artificial intelligence has'

# Compare different generation strategies
print('=== GREEDY ===')
out = generator(prompt, max_new_tokens=60, do_sample=False)
print(out[0]['generated_text'])

print('\n=== TOP-K SAMPLING (k=50, temp=0.8) ===')
out = generator(prompt, max_new_tokens=60, do_sample=True,
                top_k=50, temperature=0.8)
print(out[0]['generated_text'])

print('\n=== NUCLEUS SAMPLING (p=0.9, temp=1.0) ===')
out = generator(prompt, max_new_tokens=60, do_sample=True,
                top_p=0.9, temperature=1.0)
print(out[0]['generated_text'])
```

## 🔬 LAB: NLP Task Comparison Benchmark

In this lab, you will build a comprehensive NLP benchmark notebook that tests multiple models across multiple tasks and records performance. This gives you a template for systematically evaluating models.

```
# Lab 5: NLP Task Benchmark
from transformers import pipeline
import time

# Define benchmark inputs
benchmark = {
    'sentiment': {
        'models': ['distilbert-base-uncased-finetuned-sst-2-english',
                   'nlptown/bert-base-multilingual-uncased-sentiment'],
        'inputs': ['Great product!', 'Terrible experience.', 'It was
okay.'],
    },
    'summarization': {
        'models': ['facebook/bart-large-cnn', 't5-small'],
        'inputs': ['<long article text>'],  # Replace with actual text
    },
}

# Run benchmarks
results = {}
for task, config in benchmark.items():
    print(f'\n{"=" * 50}')
    print(f'Task: {task}')
    print(f'{"=" * 50}')
    for model_name in config['models']:
        print(f'\nModel: {model_name}')
        start = time.time()
        pipe = pipeline(task if task != 'sentiment' else 'sentiment-
analysis',
                        model=model_name, device=0)
        load_time = time.time() - start
        print(f'  Load time: {load_time:.2f}s')

        start = time.time()
        outputs = pipe(config['inputs'])
        infer_time = time.time() - start
        print(f'  Inference time: {infer_time:.3f}s')
```

```
        print(f'  Output: {outputs}')
        del pipe
        import gc; gc.collect()
        import torch; torch.cuda.empty_cache()
```

| Task | Model | Load Time | Inference Time | Quality Rating |
|---|---|---|---|---|
| Sentiment | DistilBERT-SST2 | (Fill in) | (Fill in) | (1-5) |
| Sentiment | BERT-multilingual | (Fill in) | (Fill in) | (1-5) |
| Summarization | BART-large-CNN | (Fill in) | (Fill in) | (1-5) |
| Summarization | T5-small | (Fill in) | (Fill in) | (1-5) |

*Figure: Lab 5 Benchmark Results Sheet*

## Advanced Classification: Zero-Shot and Multi-Label

Zero-shot classification is a powerful technique that allows you to classify text into categories that the model has never explicitly seen during training. Instead of fine-tuning a model on specific labels, zero-shot models use natural language inference to determine whether a text entails a given label description.

This is particularly useful in research settings where labeled data is scarce or when you need to rapidly prototype a classification system for a new domain. The trade-off is that zero-shot models are generally less accurate than fine-tuned models on specific tasks, but they require zero labeled examples.

```
from transformers import pipeline

# Zero-shot classification
zsc = pipeline('zero-shot-classification',
               model='facebook/bart-large-mnli', device=0)

text = 'The new quantum processor achieves 1000 qubits of error-corrected
computation.'
labels = ['technology', 'science', 'business', 'politics', 'sports',
'entertainment']

result = zsc(text, candidate_labels=labels, multi_label=False)
print(f'Text: {text}\n')
for label, score in zip(result['labels'], result['scores']):
    bar = '#' * int(score * 50)
    print(f'{label:>15}: {bar} ({score:.4f})')

# Multi-label classification (multiple labels can be true)
result = zsc(text, candidate_labels=labels, multi_label=True)
print(f'\nMulti-label results:')
for label, score in zip(result['labels'], result['scores']):
    if score > 0.3:  # Threshold
        print(f'  {label}: {score:.4f}')
```

## Named Entity Recognition in Detail

Named Entity Recognition (NER) identifies and classifies named entities in text into predefined categories such as person names, organizations, locations, dates, and more. It is a fundamental task in information extraction

and is widely used in research for analyzing scientific literature, news articles, and social media posts.

```
from transformers import pipeline

# NER with aggregation
ner = pipeline('ner', model='dslim/bert-base-NER',
               aggregation_strategy='simple', device=0)

text = '''
Dr. Sarah Chen from Stanford University published a groundbreaking paper
on quantum computing in Nature on March 15, 2025. Her team collaborated
with researchers from Google DeepMind and MIT to develop a new algorithm
that runs on their Willow quantum processor.
'''

entities = ner(text.strip())
print(f'Found {len(entities)} entities:\n')
for ent in entities:
    print(f'  [{ent["entity_group"]:>4}] {ent["word"]:>25} (confidence:
{ent["score"]:.3f})')
```

| Entity Type | Abbreviation | Examples | Common Models |
|---|---|---|---|
| Person | PER | Sarah Chen, Albert Einstein | bert-base-NER, xlm-roberta |
| Organization | ORG | Google, Stanford University | bert-base-NER, flair |
| Location | LOC | Paris, Mount Everest | bert-base-NER, spacy |
| Miscellaneous | MISC | English, Nobel Prize | bert-base-NER |
| Date | DATE | March 15, 2025 | Specialized NER models |
| Scientific Term | CHEM/GENE | H2O, BRCA1 | BioBERT, SciBERT |

*Figure: Common NER Entity Types and Models*

## Practical Tips for NLP Tasks

| Task | Model Selection Tip | Common Pitfall | Best Practice |
|---|---|---|---|
| Classification | Start with DistilBERT for speed | Imbalanced classes | Use weighted F1, stratified splits |

| Task | Model Selection Tip | Common Pitfall | Best Practice |
|---|---|---|---|
| Summarization | BART-CNN for news, LED for long docs | Input exceeds max length | Chunk long documents with overlap |
| Translation | MarianMT for specific pairs | Domain-specific vocabulary | Fine-tune on domain data |
| QA | RoBERTa-SQuAD for extractive | Answer not in context | Add 'unanswerable' detection |
| Generation | Start with GPT-2, scale to Mistral | Repetitive outputs | Use repetition_penalty, top_p |
| NER | bert-base-NER for general | Nested entities | Use aggregation_strategy |

*Figure: NLP Task Selection and Best Practice Guide*

# Chapter 6. Working with Embeddings and Sentence Transformers

Embeddings are dense numerical representations of text that capture semantic meaning in a vector space. This chapter explores how to generate, use, and evaluate embeddings for research applications including semantic similarity, clustering, and retrieval.

## 6.1 Understanding Embeddings

Unlike bag-of-words or TF-IDF representations, embeddings capture semantic relationships. Words and sentences with similar meanings are placed close together in the embedding space, while unrelated concepts are far apart. This property enables a wide range of applications from search to clustering.

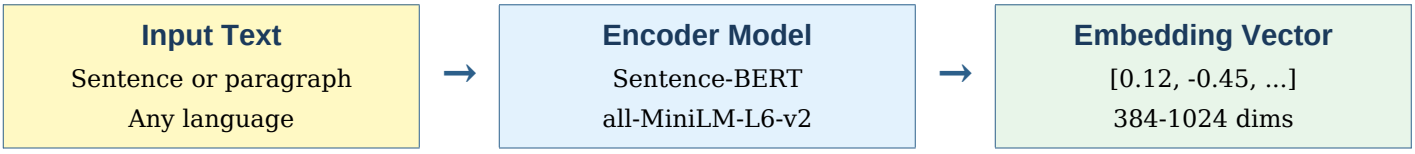


*Figure: Text to embedding conversion pipeline*

| Embedding Model | Dimensions | Max Tokens | Speed | Quality |
|---|---|---|---|---|
| all-MiniLM-L6-v2 | 384 | 256 | Very fast | Good |
| all-mpnet-base-v2 | 768 | 384 | Medium | Very good |
| bge-large-en-v1.5 | 1024 | 512 | Slow | Excellent |
| e5-large-v2 | 1024 | 512 | Slow | Excellent |
| gte-small | 384 | 512 | Very fast | Good |
| nomic-embed-text-v1.5 | 768 | 8192 | Medium | Very good |

*Figure: Popular Sentence Embedding Models*

## 6.2 Semantic Similarity

```
from sentence_transformers import SentenceTransformer, util
import torch

model = SentenceTransformer('all-MiniLM-L6-v2')

# Compute semantic similarity between sentence pairs
pairs = [
    ('The cat sat on the mat.', 'A feline rested on the rug.'),
    ('The cat sat on the mat.', 'The stock market crashed today.'),
    ('I love programming.', 'Coding is my passion.'),
    ('I love programming.', 'I enjoy cooking pasta.'),
]

for s1, s2 in pairs:
    emb1 = model.encode(s1, convert_to_tensor=True)
    emb2 = model.encode(s2, convert_to_tensor=True)
    sim = util.cos_sim(emb1, emb2).item()
    bar = '#' * int(max(0, sim) * 30)
    print(f'{sim:+.4f} {bar}')
    print(f'  "{s1}"')
    print(f'  "{s2}"\n')
```

## 6.3 Building a Semantic Search System

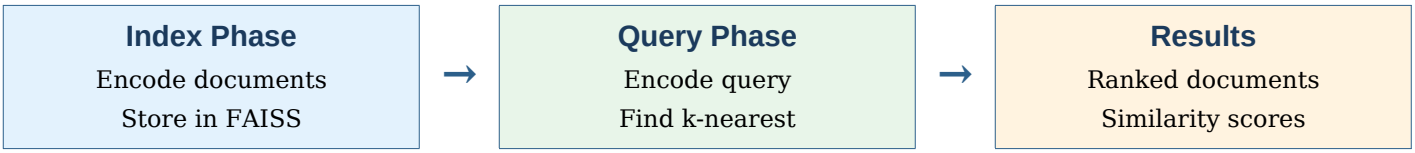


*Figure: Semantic search system architecture*

```
import faiss
import numpy as np
from sentence_transformers import SentenceTransformer

# Sample document corpus
documents = [
    'Python is a popular programming language for data science.',
    'Machine learning models require large amounts of training data.',
    'The Eiffel Tower is located in Paris, France.',
    'Deep learning uses neural networks with many layers.',
    'The Mediterranean diet is rich in olive oil and vegetables.',
    'Transfer learning allows models to reuse knowledge from pre-
training.',
    'Tokyo is the capital city of Japan.',
    'Gradient descent is an optimization algorithm for training neural
nets.',
    'The Amazon rainforest is the largest tropical forest in the world.',
    'Attention mechanisms allow models to focus on relevant input parts.',
]

# Encode and index
model = SentenceTransformer('all-MiniLM-L6-v2')
doc_embeddings = model.encode(documents, convert_to_numpy=True)

# Build FAISS index
dimension = doc_embeddings.shape[1]  # 384
index = faiss.IndexFlatIP(dimension)  # Inner product (cosine on
normalized)
faiss.normalize_L2(doc_embeddings)
index.add(doc_embeddings)

# Search
query = 'How do neural networks learn?'
query_emb = model.encode([query], convert_to_numpy=True)
faiss.normalize_L2(query_emb)
```

```
scores, indices = index.search(query_emb, k=3)

print(f'Query: "{query}"\n')
for rank, (idx, score) in enumerate(zip(indices[0], scores[0])):
    print(f'  #{rank+1} (score: {score:.4f}): {documents[idx]}')
```

## 🔬 LAB: Build a Research Paper Search Engine

Create a semantic search system that can find relevant research concepts from a corpus of AI paper abstracts. Use the provided sample abstracts or create your own.

```
# Lab 6: Research Paper Search Engine
# Follow the pattern above with 20+ paper abstracts
# Test with queries like:
#   'methods for reducing model size'
#   'how to handle imbalanced datasets'
#   'attention mechanism improvements'
# Record the top-3 results for each query
```

## Advanced Embedding Techniques

While basic sentence embeddings work well for many tasks, advanced techniques can significantly improve performance for specialized applications. Understanding these techniques will help you build more effective retrieval and similarity systems.

| Technique | Description | When to Use | Complexity |
|---|---|---|---|
| Mean Pooling | Average all token embeddings | General purpose | Simple |
| CLS Token | Use [CLS] token embedding only | BERT-style classification | Simple |
| Max Pooling | Take max across token dimension | Keyword-sensitive tasks | Simple |
| Instruction-Tuned | Prefix with task description | Multi-task embeddings | Medium |
| Matryoshka | Variable-dimension embeddings | Storage-constrained systems | Medium |
| Late Interaction | ColBERT-style token matching | High-precision retrieval | Complex |

*Figure: Advanced Embedding Techniques*

```
# Instruction-tuned embeddings for better task performance
from sentence_transformers import SentenceTransformer

model = SentenceTransformer('intfloat/e5-small-v2')

# Prefix queries and documents differently
query = 'query: How does photosynthesis work?'
docs = [
    'passage: Photosynthesis is the process by which plants convert light
energy into chemical energy.',
    'passage: The stock market experienced significant volatility this
quarter.',
    'passage: Chloroplasts contain chlorophyll that absorbs sunlight for
energy conversion.',
]

q_emb = model.encode(query)
```

```
d_embs = model.encode(docs)

from sentence_transformers import util
scores = util.cos_sim(q_emb, d_embs)[0]
for doc, score in zip(docs, scores):
    print(f'{score:.4f}: {doc[9:60]}...')
```

## Building a Complete Similarity Search Application

Let us build a more sophisticated semantic search application that includes document chunking, embedding caching, and result re-ranking. This is the kind of system you might build for searching through research papers or internal documents.

```
import numpy as np
import faiss
from sentence_transformers import SentenceTransformer, CrossEncoder
import time

# Corpus of research-related passages
corpus = [
    'Transfer learning enables models to leverage knowledge from pre-
training.',
    'Fine-tuning adapts pre-trained models to specific downstream tasks.',
    'Data augmentation increases the effective size of training
datasets.',
    'Batch normalization stabilizes training by normalizing layer
inputs.',
    'Dropout prevents overfitting by randomly deactivating neurons.',
    'Attention mechanisms allow models to focus on relevant input parts.',
    'Convolutional neural networks excel at spatial feature extraction.',
    'Recurrent networks process sequential data one step at a time.',
    'Gradient clipping prevents exploding gradients during training.',
    'Learning rate scheduling adjusts the step size during optimization.',
    'Early stopping prevents overfitting by monitoring validation loss.',
    'Cross-validation provides robust estimates of model performance.',
    'Ensemble methods combine multiple models for better predictions.',
    'Knowledge distillation transfers knowledge from large to small
models.',
    'Quantization reduces model size by lowering numerical precision.',
    'Pruning removes unnecessary weights to create smaller models.',
```

```
    'Federated learning trains models across distributed devices.',
    'Self-supervised learning creates labels from the data itself.',
    'Contrastive learning trains embeddings by comparing similar pairs.',
    'Meta-learning enables models to learn how to learn new tasks
quickly.',
]

# Step 1: Encode corpus
bi_encoder = SentenceTransformer('all-MiniLM-L6-v2')
print('Encoding corpus...')
start = time.time()
corpus_embeddings = bi_encoder.encode(corpus, convert_to_numpy=True,
                                      show_progress_bar=True)
print(f'Encoded {len(corpus)} documents in {time.time()-start:.2f}s')

# Step 2: Build FAISS index
faiss.normalize_L2(corpus_embeddings)
index = faiss.IndexFlatIP(corpus_embeddings.shape[1])
index.add(corpus_embeddings)

# Step 3: Search function with optional cross-encoder re-ranking
def search(query, top_k=5, rerank=False):
    q_emb = bi_encoder.encode([query], convert_to_numpy=True)
    faiss.normalize_L2(q_emb)
    scores, indices = index.search(q_emb, top_k)

    results = []
    for idx, score in zip(indices[0], scores[0]):
        results.append({'text': corpus[idx], 'bi_score': float(score)})

    if rerank:
        cross_encoder = CrossEncoder('cross-encoder/ms-marco-MiniLM-L-6-
v2')
        pairs = [[query, r['text']] for r in results]
        ce_scores = cross_encoder.predict(pairs)
        for r, s in zip(results, ce_scores):
            r['ce_score'] = float(s)
        results.sort(key=lambda x: x['ce_score'], reverse=True)

    return results
```

```
# Test queries
queries = [
    'How to prevent a model from overfitting?',
    'Methods to make models smaller and faster',
    'How to train with limited labeled data',
]

for query in queries:
    print(f'\nQuery: "{query}"')
    results = search(query, top_k=3)
    for i, r in enumerate(results):
        print(f'  #{i+1} ({r["bi_score"]:.4f}): {r["text"][:70]}')
```

# Chapter 7. Computer Vision with Hugging Face

Hugging Face is not just for NLP. The Transformers library provides comprehensive support for computer vision tasks including image classification, object detection, image segmentation, and image captioning. This chapter covers the main vision tasks with practical examples.

## 7.1 Image Classification

Vision Transformers (ViT) have demonstrated that the transformer architecture, originally designed for text, can achieve excellent performance on image tasks. ViT divides an image into fixed-size patches, treats each patch as a token, and processes them through standard transformer layers.

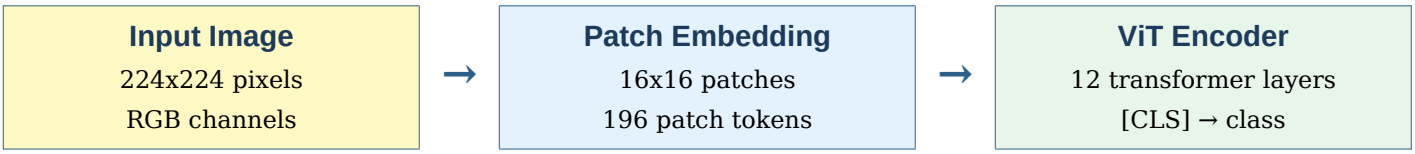


*Figure: Vision Transformer (ViT) architecture for image classification*

| Model | Top-1 Accuracy (ImageNet) | Parameters | Speed (img/s) |
|---|---|---|---|
| ViT-base-patch16-224 | 81.1% | 86M | ~850 |
| ViT-large-patch16-224 | 82.6% | 304M | ~300 |

| Model | Top-1 Accuracy (ImageNet) | Parameters | Speed (img/s) |
|---|---|---|---|
| DeiT-base | 81.8% | 86M | ~850 |
| BEiT-base | 82.9% | 86M | ~800 |
| Swin-base | 83.5% | 88M | ~750 |
| ConvNeXt-base | 83.8% | 89M | ~700 |

*Figure: Image Classification Models on Hugging Face Hub*

```
from transformers import pipeline
from PIL import Image
import requests

classifier = pipeline('image-classification',
                      model='google/vit-base-patch16-224', device=0)

# Load image from URL
url =
'https://upload.wikimedia.org/wikipedia/commons/thumb/4/4d/Cat_November_20
10-1a.jpg/1200px-Cat_November_2010-1a.jpg'
image = Image.open(requests.get(url, stream=True).raw)

results = classifier(image, top_k=5)
for r in results:
    bar = '#' * int(r['score'] * 40)
    print(f'{r["label"]:>30}: {bar} ({r["score"]:.4f})')
```

## 7.2 Object Detection

Object detection identifies and localizes multiple objects within an image. DETR (Detection Transformer) was the first fully end-to-end object detection model based on transformers, eliminating the need for hand-designed components like non-maximum suppression.

```
from transformers import pipeline

detector = pipeline('object-detection',
                    model='facebook/detr-resnet-50', device=0)
results = detector(image, threshold=0.7)
```

```
for obj in results:
    box = obj['box']
    print(f'{obj["label"]:>15} ({obj["score"]:.3f}): '
          f'x={box["xmin"]:.0f}, y={box["ymin"]:.0f}, '
          f'w={box["xmax"]-box["xmin"]:.0f}, h={box["ymax"]-
box["ymin"]:.0f}')
```

## 7.3 Image Captioning

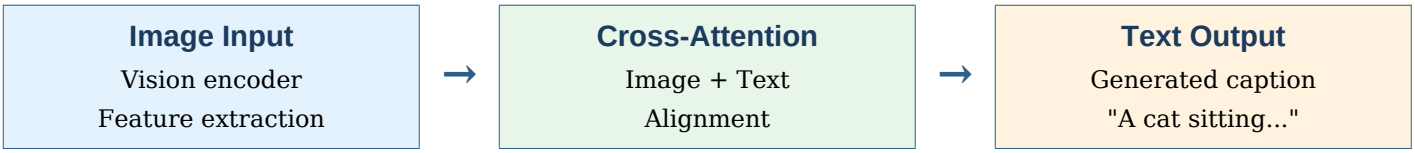


*Figure: Image captioning architecture*

```
captioner = pipeline('image-to-text',
                     model='Salesforce/blip-image-captioning-base',
device=0)
captions = captioner(image)
print(f'Caption: {captions[0]["generated_text"]}')
```

### 🔬 LAB: Computer Vision Pipeline

Build a complete vision pipeline that classifies images, detects objects, and generates captions. Test with at least 3 different images.

## Vision Transformer Architecture Deep Dive

Understanding how Vision Transformers (ViTs) work internally helps you use them more effectively and make informed decisions about model selection. Unlike convolutional neural networks that process images through sliding filters, ViTs divide the image into fixed-size patches and process them as a sequence, just like tokens in NLP.

The input image (typically 224x224 pixels) is divided into patches of size 16x16 pixels, resulting in 196 patches (14x14 grid). Each patch is flattened into a vector and projected to the model's hidden dimension through a linear embedding layer. A special classification token (similar to BERT's [CLS] token) is prepended to the sequence, and positional embeddings are added to encode spatial information.

The sequence of patch embeddings is then processed through standard transformer encoder layers with multi-head self-attention and feed-forward networks. The output corresponding to the [CLS] token is used for classification, while the full sequence output can be used for per-patch tasks like segmentation.

| Stage | Operation | Shape (ViT-base) | Purpose |
|---|---|---|---|
| Input | Raw image | 3 x 224 x 224 | RGB image |
| Patch extraction | Split into 16x16 patches | 196 x (16*16*3) | Create token-like inputs |
| Patch embedding | Linear projection | 196 x 768 | Project to hidden dim |
| Add [CLS] token | Prepend learnable token | 197 x 768 | Classification anchor |
| Position embedding | Add positional info | 197 x 768 | Encode spatial positions |
| Transformer layers | 12 layers of MSA + FFN | 197 x 768 | Learn representations |
| Classification head | Linear on [CLS] output | 1 x num_classes | Final prediction |

*Figure: Vision Transformer Processing Pipeline: Stage by Stage*

## Image Segmentation

Image segmentation goes beyond classification and detection by assigning a class label to every pixel in the image. This is crucial for applications like medical imaging, autonomous driving, and satellite imagery analysis. Hugging Face supports several segmentation architectures.

| Segmentation Type | Description | Models on Hub | Example Application |
|---|---|---|---|
| Semantic | Label every pixel by class | SegFormer, Mask2Former | Land use mapping |
| Instance | Separate individual objects | Mask R-CNN, DETR | Cell counting in microscopy |
| Panoptic | Semantic + Instance combined | Mask2Former, OneFormer | Autonomous driving |

*Figure: Types of Image Segmentation*

```
from transformers import pipeline

# Semantic segmentation
segmenter = pipeline('image-segmentation',
                     model='nvidia/segformer-b0-finetuned-ade-512-512',
                     device=0)

result = segmenter('path/to/image.jpg')
for segment in result:
    print(f'  {segment["label"]:>20}: score={segment["score"]:.3f}')
```

## Multimodal Models: Vision + Language

Multimodal models that combine vision and language capabilities represent one of the most exciting frontiers in AI research. These models can understand images in the context of text, enabling tasks like visual question answering, image captioning, and text-guided image retrieval.

| Model | Capabilities | Architecture | Parameters | Colab Compatible? |
|---|---|---|---|---|
| CLIP | Image-text matching, zero-shot classification | Dual encoder | 400M | Yes (Free tier) |
| BLIP-2 | Image captioning, VQA | Frozen encoder + Q-Former | 3.4B | Yes (quantized) |
| LLaVA | Visual conversation, reasoning | ViT + LLaMA | 7-13B | QLoRA only |
| Florence-2 | Multiple vision tasks | Encoder-decoder | 232M | Yes (Free tier) |

| Model | Capabilities | Architecture | Parameters | Colab Compatible? |
|---|---|---|---|---|
| InternVL | General vision-language | ViT + LLM | Various | Depends on size |

*Figure: Multimodal Models Available on Hugging Face Hub*

```
# CLIP: Zero-shot image classification
from transformers import CLIPProcessor, CLIPModel
from PIL import Image
import requests

model = CLIPModel.from_pretrained('openai/clip-vit-base-patch32')
processor = CLIPProcessor.from_pretrained('openai/clip-vit-base-patch32')

# Load an image
url =
'https://upload.wikimedia.org/wikipedia/commons/thumb/4/4d/Cat_November_20
10-1a.jpg/1200px-Cat_November_2010-1a.jpg'
image = Image.open(requests.get(url, stream=True).raw)

# Zero-shot classification with custom labels
labels = ['a photo of a cat', 'a photo of a dog', 'a photo of a bird',
          'a photo of a car', 'a photo of a house']

inputs = processor(text=labels, images=image, return_tensors='pt',
padding=True)
outputs = model(**inputs)
probs = outputs.logits_per_image.softmax(dim=1)[0]

print('Zero-shot classification results:')
for label, prob in sorted(zip(labels, probs), key=lambda x: -x[1]):
    bar = '#' * int(prob * 40)
    print(f'  {label:>25}: {bar} ({prob:.4f})')
```

## 🔬 LAB: Extended Computer Vision Lab: Multi-Task Pipeline

Build a comprehensive computer vision pipeline that performs classification, detection, captioning, and segmentation on the same image. Compare the results and analyze the strengths of each model.

```
# Extended CV Lab
from transformers import pipeline
from PIL import Image
import requests, time

# Load test image
url =
'https://upload.wikimedia.org/wikipedia/commons/4/4d/Cat_November_2010-
1a.jpg'
image = Image.open(requests.get(url, stream=True).raw)

tasks = {
    'classification': pipeline('image-classification', device=0),
    'detection': pipeline('object-detection',
                          model='facebook/detr-resnet-50', device=0),
    'captioning': pipeline('image-to-text',
                           model='Salesforce/blip-image-captioning-base',
device=0),
}

print('Multi-Task Vision Analysis')
print('=' * 50)
for task_name, pipe in tasks.items():
    start = time.time()
    result = pipe(image)
    elapsed = time.time() - start
    print(f'\n{task_name.upper()} ({elapsed:.2f}s):')
    if task_name == 'classification':
        for r in result[:3]:
            print(f'  {r["label"]}: {r["score"]:.4f}')
    elif task_name == 'detection':
        for r in result:
            if r['score'] > 0.5:
                print(f'  {r["label"]}: {r["score"]:.4f} at {r["box"]}')
    elif task_name == 'captioning':
        print(f'  Caption: {result[0]["generated_text"]}')
```

| Task | Model Used | Inference Time | Top Result | Quality (1-5) |
|---|---|---|---|---|
| Classification | ViT-base | (Record) | (Record) | (Rate) |

| Task | Model Used | Inference Time | Top Result | Quality (1-5) |
|---|---|---|---|---|
| Object Detection | DETR-ResNet50 | (Record) | (Record) | (Rate) |
| Image Captioning | BLIP-base | (Record) | (Record) | (Rate) |

*Figure: Lab Results Sheet: Multi-Task Vision Analysis*

# Chapter 8. Audio and Speech Models

The Hugging Face ecosystem supports a growing range of audio and speech models. This chapter covers the three main audio tasks: speech recognition, audio classification, and speech generation.

## 8.1 Speech Recognition (ASR)

Automatic speech recognition converts spoken language into text. OpenAI's Whisper is the most popular ASR model on the Hub, supporting over 90 languages with strong performance even on noisy audio.

| Whisper Model | Parameters | Relative Speed | English WER | VRAM Usage |
|---|---|---|---|---|
| whisper-tiny | 39M | 32x | ~8.0% | ~1 GB |
| whisper-base | 74M | 16x | ~5.7% | ~1.5 GB |
| whisper-small | 244M | 6x | ~4.4% | ~2.5 GB |
| whisper-medium | 769M | 2x | ~3.5% | ~5 GB |
| whisper-large-v3 | 1550M | 1x | ~2.5% | ~10 GB |

*Figure: Whisper Model Sizes and Performance*

```
from transformers import pipeline

# Load Whisper ASR pipeline
asr = pipeline('automatic-speech-recognition',
               model='openai/whisper-base', device=0)

# Transcribe audio file
result = asr('audio_sample.wav')
print(f'Transcription: {result["text"]}')

# For long audio with timestamps
result = asr('long_audio.wav',
             return_timestamps=True,
             chunk_length_s=30)
for chunk in result['chunks']:
    start, end = chunk['timestamp']
    print(f'[{start:.1f}s - {end:.1f}s] {chunk["text"]}')
```

## 8.2 Audio Classification

Audio classification assigns labels to audio clips for tasks like music genre detection, environmental sound recognition, emotion detection in speech, and speaker identification.

## 8.3 Speech Generation (TTS)

Text-to-speech models generate natural-sounding audio from text input. Models like SpeechT5 and Bark produce increasingly realistic speech output.

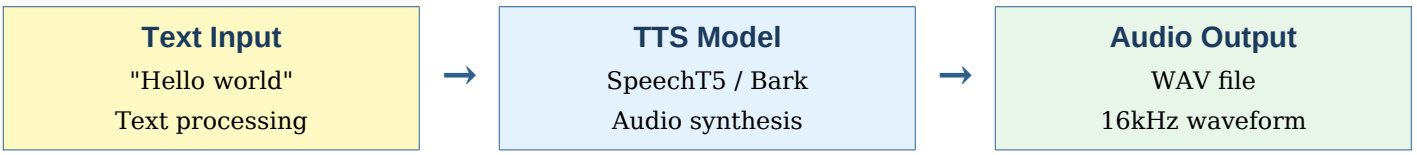


*Figure: Text-to-speech pipeline*

### 🔬 LAB: Audio Processing Pipeline

Build a pipeline that transcribes audio, classifies the emotion/content, and generates a summary. Test with different audio samples.

## Working with Audio Data in Colab

Audio processing in Colab requires some additional setup compared to text and image tasks. Audio files need to be loaded, resampled to the correct sample rate, and potentially converted between formats. The librosa and torchaudio libraries provide the necessary tools.

```
# Audio data handling in Colab
!pip install -q librosa soundfile

import librosa
import numpy as np
import IPython.display as ipd

# Load audio file
audio, sr = librosa.load('sample.wav', sr=16000)  # Resample to 16kHz
print(f'Audio shape: {audio.shape}')
print(f'Sample rate: {sr} Hz')
print(f'Duration: {len(audio)/sr:.2f} seconds')

# Play audio in notebook
ipd.Audio(audio, rate=sr)

# Basic audio analysis
print(f'Min amplitude: {audio.min():.4f}')
print(f'Max amplitude: {audio.max():.4f}')
print(f'RMS energy: {np.sqrt(np.mean(audio**2)):.4f}')
```

## Whisper: Advanced Usage

Whisper supports several advanced features beyond basic transcription. You can use it for language detection, translation (any language to English), word-level timestamps, and voice activity detection.

```
from transformers import pipeline

# Advanced Whisper usage
asr = pipeline('automatic-speech-recognition',
               model='openai/whisper-small',
               device=0)
```

```
# Basic transcription
result = asr('audio.wav')
print(f'Transcription: {result["text"]}')

# With timestamps
result = asr('audio.wav', return_timestamps=True)
for chunk in result['chunks']:
    t0, t1 = chunk['timestamp']
    print(f'  [{t0:>6.1f}s - {t1:>6.1f}s]: {chunk["text"]}')

# Translate to English (from any language)
result = asr('foreign_language_audio.wav',
             generate_kwargs={'task': 'translate'})
print(f'Translation: {result["text"]}')

# Specify source language for better accuracy
result = asr('japanese_audio.wav',
             generate_kwargs={'language': 'japanese'})
```

| Feature | Parameter | Example |
|---|---|---|
| Transcription | Default behavior | asr('audio.wav') |
| Timestamps | return_timestamps=True | Word/chunk-level timing |
| Translation | task='translate' | Any language → English |
| Language hint | language='japanese' | Improve accuracy for known language |
| Long audio | chunk_length_s=30 | Process in 30-second chunks |
| Return language | return_language=True | Detect spoken language |

*Figure: Whisper Advanced Features Reference*

## Audio Classification in Detail

Audio classification assigns labels to audio clips. The typical approach involves converting the raw audio waveform into a spectrogram or mel-spectrogram, which can then be processed by a transformer model similar to how image classifiers process images.

```
from transformers import pipeline

# Audio classification
```

```
audio_classifier = pipeline('audio-classification',
    model='MIT/ast-finetuned-audioset-10-10-0.4593', device=0)

result = audio_classifier('audio_clip.wav', top_k=5)
print('Audio Classification Results:')
for r in result:
    bar = '#' * int(r['score'] * 40)
    print(f'  {r["label"]:>30}: {bar} ({r["score"]:.4f})')
```

| Application | Model | Classes | Dataset |
|---|---|---|---|
| Environmental sounds | AST (AudioSet) | 527 sound types | AudioSet |
| Music genre | wav2vec2-music-genre | 10 genres | GTZAN |
| Speech emotion | wav2vec2-emotion | 7 emotions | RAVDESS |
| Speaker ID | wavlm-speaker-verify | Per-speaker | VoxCeleb |
| Language ID | wav2vec2-lang-id | 107 languages | Common Voice |

*Figure: Audio Classification Applications and Models*

## 🔬 LAB: Complete Audio Processing Pipeline

Build an audio analysis pipeline that transcribes speech, classifies the audio content, and detects the language. Use at least two different audio samples.

```
# Lab: Complete Audio Pipeline
from transformers import pipeline
from datasets import load_dataset

# Load sample audio from a dataset
ds = load_dataset('mozilla-foundation/common_voice_11_0',
                  'en', split='test[:5]', trust_remote_code=True)

# Initialize pipelines
asr = pipeline('automatic-speech-recognition',
               model='openai/whisper-base', device=0)
classifier = pipeline('audio-classification', device=0)

# Process each sample
```

```
for i, sample in enumerate(ds):
    audio = sample['audio']
    print(f'\nSample {i+1}:')
    print(f'  Duration:
{len(audio["array"])/audio["sampling_rate"]:.1f}s')

    # Transcribe
    transcript = asr(audio['array'], sampling_rate=audio['sampling_rate'])
    print(f'  Transcript: {transcript["text"]}')

    # Classify
    classes = classifier(audio['array'],
sampling_rate=audio['sampling_rate'], top_k=3)
    print(f'  Classification:')
    for c in classes:
        print(f'    {c["label"]}: {c["score"]:.4f}')
```

# Part II Capstone Lab

## 🔬 LAB: Multi-Modal AI Research Assistant

In this capstone lab, you will build a multi-modal research assistant that can analyze text, images, and audio. This integrates all the skills from Chapters 4 through 8 into a single cohesive application.

**Objective:** Build a pipeline that accepts text, image URLs, or audio files and performs appropriate analysis based on the input modality. The system should detect the input type and route it to the correct model pipeline.

**Duration:** 90–120 minutes

### Architecture Design

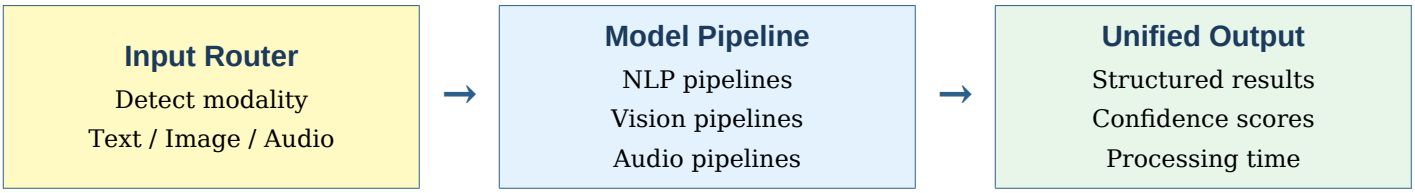


*Figure: Multi-modal research assistant architecture*

### Implementation

```
from transformers import pipeline
from PIL import Image
import requests, time

class MultiModalAssistant:
    def __init__(self, device=0):
        print('Loading models...')
        self.device = device
        self.models = {}
        self._load_text_models()
        self._load_vision_models()
        print('All models loaded!')

    def _load_text_models(self):
```

```
        self.models['sentiment'] = pipeline('sentiment-analysis',
device=self.device)
        self.models['ner'] = pipeline('ner',
aggregation_strategy='simple', device=self.device)
        self.models['summarize'] = pipeline('summarization',
            model='facebook/bart-large-cnn', device=self.device)
        self.models['qa'] = pipeline('question-answering',
device=self.device)
        self.models['zsc'] = pipeline('zero-shot-classification',
device=self.device)
        print('  Text models: 5 loaded')

    def _load_vision_models(self):
        self.models['img_clf'] = pipeline('image-classification',
device=self.device)
        self.models['caption'] = pipeline('image-to-text',
            model='Salesforce/blip-image-captioning-base',
device=self.device)
        print('  Vision models: 2 loaded')

    def analyze_text(self, text, tasks=None):
        if tasks is None:
            tasks = ['sentiment', 'ner']
        results = {'modality': 'text', 'input_preview': text[:100]}
        for task in tasks:
            start = time.time()
            if task == 'sentiment':
                r = self.models['sentiment'](text[:512])[0]
                results['sentiment'] = {'label': r['label'], 'score':
round(r['score'], 4)}
            elif task == 'ner':
                entities = self.models['ner'](text[:512])
                results['entities'] = [{'text': e['word'], 'type':
e['entity_group'],
                    'score': round(e['score'], 3)} for e in entities]
            elif task == 'summarize' and len(text.split()) > 40:
                r = self.models['summarize'](text[:1024], max_length=100)
[0]
                results['summary'] = r['summary_text']
            elif task == 'classify':
                labels = ['technology', 'science', 'business', 'politics',
'health']
```

```
                r = self.models['zsc'](text[:512],
candidate_labels=labels)
                results['topics'] = dict(zip(r['labels'][:3],
                    [round(s, 3) for s in r['scores'][:3]]))
            results[f'{task}_time'] = round(time.time() - start, 3)
        return results

    def analyze_image(self, image_source):
        if isinstance(image_source, str) and
image_source.startswith('http'):
            image = Image.open(requests.get(image_source,
stream=True).raw)
        else:
            image = image_source
        results = {'modality': 'image'}
        start = time.time()
        clf = self.models['img_clf'](image, top_k=5)
        results['classification'] = [{'label': r['label'], 'score':
round(r['score'], 4)}
            for r in clf]
        results['clf_time'] = round(time.time() - start, 3)
        start = time.time()
        cap = self.models['caption'](image)
        results['caption'] = cap[0]['generated_text']
        results['caption_time'] = round(time.time() - start, 3)
        return results

    def print_results(self, results):
        print('\n' + '=' * 60)
        print(f'ANALYSIS RESULTS ({results["modality"].upper()})')
        print('=' * 60)
        for key, val in results.items():
            if key.endswith('_time'):
                continue
            if key in ['modality', 'input_preview']:
                continue
            print(f'\n{key.upper()}:')
            if isinstance(val, list):
                for item in val[:5]:
                    print(f'  {item}')
            elif isinstance(val, dict):
```

```
                for k, v in val.items():
                    print(f'  {k}: {v}')
            else:
                print(f'  {val}')
```

## Testing the Assistant

```
# Initialize
assistant = MultiModalAssistant(device=0)

# Test 1: Text Analysis
text = '''
Researchers at Stanford University have developed a new AI system that can
predict protein structures with unprecedented accuracy. The breakthrough,
published in Nature on March 10, 2025, could accelerate drug discovery
by reducing the time needed to understand protein interactions from months
to hours. Dr. Jennifer Park, lead author of the study, said the system
achieves 95% accuracy on standard benchmarks, surpassing previous methods
by a significant margin.
'''
results = assistant.analyze_text(text, tasks=['sentiment', 'ner',
'summarize', 'classify'])
assistant.print_results(results)

# Test 2: Image Analysis
url =
'https://upload.wikimedia.org/wikipedia/commons/4/4d/Cat_November_2010-
1a.jpg'
results = assistant.analyze_image(url)
assistant.print_results(results)
```

## Lab Report

| Test | Input Type | Tasks Run | Total Time | Quality Rating |
|---|---|---|---|---|
| 1: News article | Text (150 words) | sentiment, NER, summary, classify | (Record) | (1-5) |
| 2: Scientific abstract | Text (200 words) | sentiment, NER, summary, classify | (Record) | (1-5) |
| 3: Cat photo | Image (URL) | classification, captioning | (Record) | (1-5) |

| Test | Input Type | Tasks Run | Total Time | Quality Rating |
|---|---|---|---|---|
| 4: Landscape photo | Image (URL) | classification, captioning | (Record) | (1-5) |
| 5: Your own text | Text | All available tasks | (Record) | (1-5) |

*Figure: Part II Capstone Lab Report Sheet*

| Reflection Question | Your Answer |
|---|---|
| Which NLP task was most accurate on your inputs? | (Write) |
| Which task was slowest? Why? | (Write) |
| How well did zero-shot classification work vs specialized models? | (Write) |
| What improvements would you make to this assistant? | (Write) |
| How would you extend this to support audio inputs? | (Write) |

*Figure: Part II Capstone Reflection Questions*

# PART III

## DATA AND EXPERIMENTATION

# Chapter 9. Using Hugging Face Datasets in Colab

Data is the foundation of every machine learning project. This chapter provides a comprehensive guide to using the Hugging Face Datasets library in Google Colab, covering everything from basic loading to advanced streaming techniques for handling datasets that exceed Colab's memory limits.

## 9.1 Loading Datasets

```
from datasets import load_dataset

# Load from Hub with automatic caching
dataset = load_dataset('imdb')
print(dataset)
print(f'Train examples: {len(dataset["train"]):,}')
print(f'Test examples: {len(dataset["test"]):,}')
print(f'Features: {dataset["train"].features}')
print(f'First example: {dataset["train"][0]}')

# Load specific split
train = load_dataset('imdb', split='train')

# Load specific subset with percentage
small = load_dataset('imdb', split='train[:10%]')
```

## 9.2 Working with Splits

```
# Create custom train/val split from training data
split_dataset = dataset['train'].train_test_split(test_size=0.1, seed=42)
print(f'New train: {len(split_dataset["train"]):,}')
print(f'Validation: {len(split_dataset["test"]):,}')
```

## 9.3 Filtering, Mapping, and Sampling

| Operation | Code Pattern | Use Case |
|---|---|---|
| Filter | .filter(lambda x: x['label'] == 1) | Select subset by condition |
| Map | .map(tokenize_fn, batched=True) | Transform all examples |
| Select | .select(range(1000)) | Choose by index |
| Shuffle | .shuffle(seed=42) | Randomize order |
| Sort | .sort('length') | Order by column |
| Rename | .rename_column('text', 'input') | Rename columns |
| Remove | .remove_columns(['unused']) | Drop columns |
| Flatten | .flatten() | Unnest dictionaries |

*Figure: Common Dataset Operations Quick Reference*

## 9.4 Streaming Large Datasets

```
# Streaming mode for datasets too large for memory
stream = load_dataset('c4', 'en', split='train', streaming=True)

# Take first 100 examples
for i, example in enumerate(stream):
    if i >= 100: break
    print(f'[{i}] {example["text"][:80]}...')

# Apply transformations lazily
filtered = stream.filter(lambda x: len(x['text']) > 500)
mapped = filtered.map(lambda x: {'length': len(x['text'])})
```

### 🔬 LAB: Dataset Exploration and Preprocessing Pipeline

```
# Lab 9: Complete dataset preprocessing
from datasets import load_dataset
from transformers import AutoTokenizer

# Load and explore
ds = load_dataset('ag_news')
print(f'Classes: {ds["train"].features["label"].names}')
print(f'Train size: {len(ds["train"]):,}')

# Analyze class distribution
```

```
from collections import Counter
dist = Counter(ds['train']['label'])
for label_id, count in sorted(dist.items()):
    name = ds['train'].features['label'].names[label_id]
    bar = '#' * (count // 1000)
    print(f'  {name:>10}: {bar} ({count:,})')

# Tokenize
tokenizer = AutoTokenizer.from_pretrained('distilbert-base-uncased')
def tokenize(batch):
    return tokenizer(batch['text'], truncation=True, padding=True)

tokenized = ds.map(tokenize, batched=True, batch_size=1000)
tokenized = tokenized.remove_columns(['text'])
tokenized.set_format('torch')
print(f'Tokenized features: {tokenized["train"].column_names}')
```

## Advanced Dataset Operations

The Datasets library supports advanced operations that are essential for sophisticated data pipelines. Understanding these operations will help you handle complex data preprocessing scenarios that arise in real research projects.

```
from datasets import load_dataset, concatenate_datasets,
interleave_datasets

# Load and combine multiple datasets
imdb = load_dataset('imdb', split='train[:1000]')
yelp = load_dataset('yelp_polarity', split='train[:1000]')

# Rename columns to match
yelp = yelp.rename_column('text', 'text')

# Concatenate datasets
combined = concatenate_datasets([imdb, yelp])
print(f'Combined: {len(combined)} examples')

# Interleave with alternating strategy
interleaved = interleave_datasets([imdb, yelp])
print(f'Interleaved: {len(list(interleaved.take(10)))} first examples')
```

## Dataset Statistics and Profiling

Before using any dataset, it is important to understand its characteristics. Profiling your data helps you make informed decisions about preprocessing, model selection, and evaluation strategies.

```
from datasets import load_dataset
import numpy as np
from collections import Counter

dataset = load_dataset('ag_news', split='train')

# Basic statistics
print(f'Total examples: {len(dataset):,}')
print(f'Features: {list(dataset.features.keys())}')
print(f'Label names: {dataset.features["label"].names}')
```

```
# Class distribution
print(f'\nClass Distribution:')
dist = Counter(dataset['label'])
for label_id, count in sorted(dist.items()):
    name = dataset.features['label'].names[label_id]
    pct = count / len(dataset) * 100
    bar = '#' * int(pct)
    print(f'  {name:>12}: {bar} {count:>6,} ({pct:.1f}%)')

# Text length statistics
lengths = [len(text.split()) for text in dataset['text']]
print(f'\nText Length (words):')
print(f'  Min:    {min(lengths):,}')
print(f'  Max:    {max(lengths):,}')
print(f'  Mean:   {np.mean(lengths):,.0f}')
print(f'  Median: {np.median(lengths):,.0f}')
print(f'  Std:    {np.std(lengths):,.0f}')

# Token length statistics (important for choosing max_length)
from transformers import AutoTokenizer
tokenizer = AutoTokenizer.from_pretrained('distilbert-base-uncased')
token_lengths = [len(tokenizer.encode(text)) for text in dataset['text']
[:5000]]
print(f'\nToken Lengths (first 5000 examples):')
print(f'  Mean:   {np.mean(token_lengths):.0f}')
print(f'  95th percentile: {np.percentile(token_lengths, 95):.0f}')
print(f'  99th percentile: {np.percentile(token_lengths, 99):.0f}')
print(f'  -> Recommended max_length: {int(np.percentile(token_lengths,
95))}')
```

| Statistic | Why It Matters | What to Do If Unusual |
|---|---|---|
| Class distribution | Imbalanced classes need special handling | Use weighted loss, oversample minority |
| Text length range | Determines max_length setting | Set to 95th-99th percentile |
| Missing values | Can crash preprocessing | Filter or impute before training |
| Duplicate examples | Inflates metrics artificially | Deduplicate, especially across splits |
| Language | Multilingual data needs | Filter or use multilingual model |

| Statistic | Why It Matters | What to Do If Unusual |
|---|---|---|
| distribution | appropriate models | |
| Special characters | May affect tokenization | Clean or normalize as needed |

*Figure: Dataset Profiling Checklist*

# Chapter 10. Tokenization, Preprocessing, and Data Collation

Proper preprocessing is the difference between a model that works and one that fails silently. This chapter covers the critical details of tokenization, padding, truncation, and batch preparation that are often overlooked but essential for successful training.

## 10.1 Padding Strategies

| Strategy | Code | Memory | Speed | When to Use |
|---|---|---|---|---|
| Max length | padding='max_length' | High (wastes on short) | Fast | Fixed-length pipelines |
| Longest in batch | padding='longest' | Medium | Medium | Training with collator |
| No padding | padding=False | Minimal | N/A | Single examples |
| Dynamic (collator) | DataCollatorWithPadding | Optimal | Optimal | Best practice for training |

*Figure: Padding Strategy Comparison*

## 10.2 Truncation

```
# Truncation strategies
tokenizer = AutoTokenizer.from_pretrained('bert-base-uncased')

long_text = 'word ' * 1000  # 1000 words

# Right truncation (default, keeps beginning)
```

```
enc = tokenizer(long_text, truncation=True, max_length=512)
print(f'Right truncation: {len(enc["input_ids"])} tokens')

# With stride for overlapping chunks
enc = tokenizer(long_text, truncation=True, max_length=512,
                return_overflowing_tokens=True, stride=128)
print(f'Chunks with stride: {len(enc["input_ids"])} chunks')
```

## 10.3 Data Collators

```
from transformers import DataCollatorWithPadding,
DataCollatorForLanguageModeling

# For classification: dynamic padding
data_collator = DataCollatorWithPadding(tokenizer=tokenizer)

# For masked language modeling
mlm_collator = DataCollatorForLanguageModeling(
    tokenizer=tokenizer, mlm=True, mlm_probability=0.15)

# For causal language modeling
clm_collator = DataCollatorForLanguageModeling(
    tokenizer=tokenizer, mlm=False)
```

## 10.4 Common Preprocessing Mistakes

| Mistake | Symptom | Fix |
|---|---|---|
| Wrong tokenizer for model | Garbage outputs | Always use matching tokenizer |
| Missing truncation | Runtime error on long inputs | Set truncation=True |
| Inconsistent max_length | Shape mismatch errors | Use same max_length everywhere |
| Wrong label format | Loss is NaN | Check model expects int vs float |
| Unused columns in dataset | Trainer error | Use remove_columns in .map() |
| Forgetting attention mask | Model ignores padding | Always pass attention_mask |

| Mistake | Symptom | Fix |
|---|---|---|
| Not setting format | Slow data loading | Call .set_format('torch') |

*Figure: Common Preprocessing Mistakes and Their Fixes*

## 🔬 LAB: Preprocessing Pipeline for Text Classification

Build a complete preprocessing pipeline for the AG News dataset that handles tokenization, padding, truncation, and data collation correctly.

## Advanced Tokenization Patterns

Beyond basic tokenization, there are several advanced patterns that researchers frequently need. These include handling multiple inputs (for tasks like sentence pair classification), dealing with special characters and Unicode, tokenizing structured data, and managing very long documents.

```
from transformers import AutoTokenizer

tokenizer = AutoTokenizer.from_pretrained('bert-base-uncased')

# Pattern 1: Sentence pairs (e.g., NLI, paraphrase detection)
enc = tokenizer(
    'What is machine learning?',  # sentence A
    'ML is a subset of AI.',       # sentence B
    return_tensors='pt', truncation=True, max_length=128
)
print(f'Sentence pair encoding:')
print(f'  Token type IDs: {enc["token_type_ids"][0].tolist()}')
print(f'  (0 = sentence A, 1 = sentence B)')

# Pattern 2: Batch encoding with padding
texts = ['Short text.', 'A slightly longer piece of text.', 'Word.']
batch = tokenizer(texts, return_tensors='pt',
                  padding=True, truncation=True, max_length=32)
print(f'\nBatch encoding shapes:')
print(f'  Input IDs: {batch["input_ids"].shape}')
print(f'  Attention mask: {batch["attention_mask"].shape}')
for i, (ids, mask) in enumerate(zip(batch['input_ids'],
batch['attention_mask'])):
    real_tokens = mask.sum().item()
    total = len(ids)
    print(f'  Text {i}: {real_tokens} real tokens, {total-real_tokens}
padding tokens')

# Pattern 3: Handling long documents with stride
long_text = 'This is a very long document. ' * 200
chunks = tokenizer(
    long_text, return_tensors='pt', truncation=True,
    max_length=128, return_overflowing_tokens=True, stride=32
)
```

```
print(f'\nLong document chunking:')
print(f'  Original length: ~{len(long_text.split())} words')
print(f'  Number of chunks: {len(chunks["input_ids"])}')
print(f'  Chunk size: {chunks["input_ids"][0].shape[0]} tokens')
print(f'  Overlap: 32 tokens between consecutive chunks')
```

## The Complete Preprocessing Pipeline: A Visual Guide

Understanding the complete preprocessing pipeline from raw data to model-ready batches is essential for building robust training workflows. Here is a visual representation of each stage and the decisions you need to make at each step.

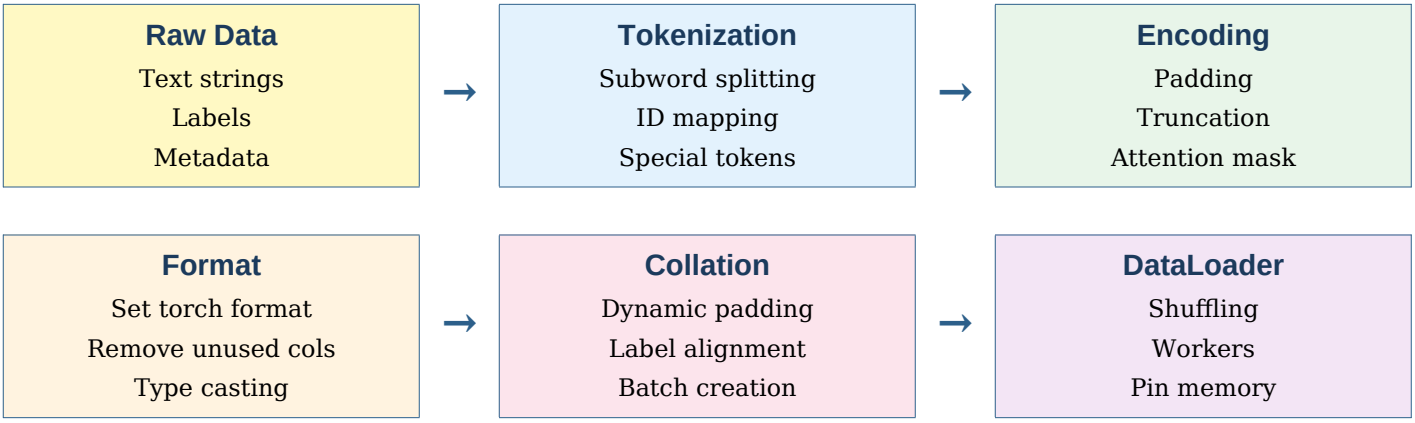


*Figure: Complete preprocessing pipeline: from raw data to training-ready DataLoader*

| Stage | Key Decision | Options | Recommendation |
|---|---|---|---|
| Tokenization | Which tokenizer? | Always match the model | Use AutoTokenizer |
| Max length | How long? | 128, 256, 512, 1024 | Use 95th percentile of text lengths |
| Padding | When to pad? | batch, max_length, none | Use DataCollatorWithPadding |
| Truncation | Which side? | right (default), left | Right for classification, left for QA |
| Batching | Batch size? | 8, 16, 32, 64 | Largest that fits in GPU memory |
| Format | Output type? | torch, numpy, jax | Match your framework |

*Figure: Preprocessing Decision Guide*

# Chapter 11. Experiment Tracking and Reproducibility

Reproducibility is a cornerstone of good research. This chapter covers the tools and practices that ensure your experiments can be replicated and your results are trustworthy.

## 11.1 Setting Random Seeds

```
from transformers import set_seed
import torch, random, numpy as np

def set_all_seeds(seed=42):
    set_seed(seed)  # Handles torch, numpy, random
    torch.backends.cudnn.deterministic = True
    torch.backends.cudnn.benchmark = False
    print(f'All random seeds set to {seed}')

set_all_seeds(42)
```

## 11.2 Notebook Hygiene Best Practices

| Practice | Description | Priority |
|---|---|---|
| Run All before sharing | Verify top-to-bottom execution | Critical |
| Pin library versions | Record exact versions used | Critical |
| Document assumptions | Explain why, not just what | High |
| Use text cells liberally | Create narrative flow | High |
| Clear outputs before sharing | Reduce file size | Medium |
| Number experiments | exp_001, exp_002, etc. | Medium |
| Separate setup from logic | First N cells = environment | Medium |

*Figure: Notebook Hygiene Checklist*

## 11.3 Experiment Logging

```
# Using TensorBoard in Colab
%load_ext tensorboard

from transformers import TrainingArguments
training_args = TrainingArguments(
    output_dir='./results',
    logging_dir='./logs',
    logging_steps=10,
    report_to='tensorboard',
)

# Launch TensorBoard
%tensorboard --logdir ./logs
```

## 11.4 Saving Artifacts to Google Drive

```
import json, os

# Save experiment config
config = {
    'model': 'bert-base-uncased',
    'learning_rate': 2e-5,
    'batch_size': 16,
    'epochs': 3,
    'seed': 42,
    'max_length': 512,
}

save_dir = '/content/drive/MyDrive/hf_research/exp_001'
os.makedirs(save_dir, exist_ok=True)
with open(f'{save_dir}/config.json', 'w') as f:
    json.dump(config, f, indent=2)
print(f'Config saved to {save_dir}')
```

## 11.5 Reproducible Workflow Template

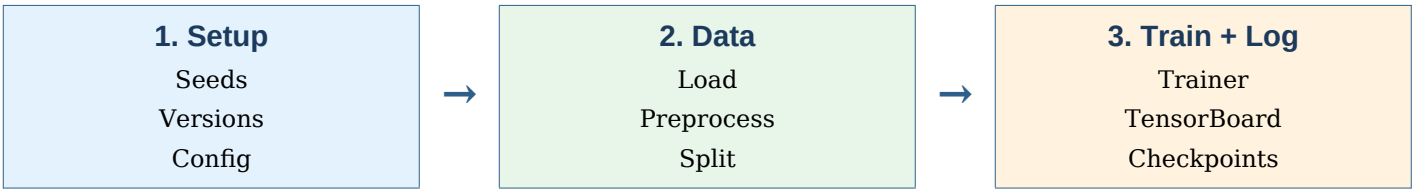


*Figure: Reproducible experiment workflow*

## 🔬 LAB: Reproducibility Audit

Take a previous experiment notebook and audit it for reproducibility. Add seeds, version pinning, configuration logging, and artifact saving.

## Version Control for ML Experiments

Traditional software version control with Git works well for code, but machine learning experiments involve additional artifacts like model weights, datasets, configurations, and results that require special handling. Here are strategies for managing these artifacts effectively.

| Artifact | Storage Strategy | Tool | Best Practice |
|---|---|---|---|
| Code | Git repository | GitHub, GitLab | Commit every experiment change |
| Model weights | Model registry | HF Hub, DVC | Tag with experiment ID |
| Datasets | Versioned storage | HF Datasets, DVC | Never modify original splits |
| Configs | Git + JSON/YAML | Hydra, OmegaConf | One config file per experiment |
| Results | Structured logging | W&B, TensorBoard | Log all metrics automatically |
| Notebooks | Git + nbstripout | GitHub, Colab | Clear outputs before commit |
| Environment | Requirements file | pip freeze | Pin exact versions |

*Figure: ML Experiment Artifact Management Guide*

## Systematic Hyperparameter Search

Tuning hyperparameters is one of the most time-consuming aspects of machine learning research. A systematic approach saves time and produces better results than random exploration.

```
# Simple grid search in Colab
import itertools
from transformers import TrainingArguments

# Define search space
param_grid = {
    'learning_rate': [1e-5, 2e-5, 5e-5],
    'per_device_train_batch_size': [16, 32],
    'num_train_epochs': [3, 5],
    'weight_decay': [0.0, 0.01],
}

# Generate all combinations
```

```
keys = param_grid.keys()
values = param_grid.values()
experiments = [dict(zip(keys, v)) for v in itertools.product(*values)]
print(f'Total experiments: {len(experiments)}')

# Run experiments (simplified)
results = []
for i, params in enumerate(experiments):
    print(f'\nExperiment {i+1}/{len(experiments)}: {params}')

    training_args = TrainingArguments(
        output_dir=f'./exp_{i:03d}',
        **params,
        evaluation_strategy='epoch',
        fp16=True,
        logging_steps=50,
        report_to='none',
    )
    # ... create trainer and train ...
    # results.append({'params': params, 'accuracy': accuracy})

# Find best
# best = max(results, key=lambda x: x['accuracy'])
# print(f'Best: {best}')
```

| Search Method | Experiments Needed | Quality | When to Use |
|---|---|---|---|
| Grid Search | All combinations | Thorough | Small search space (< 50 combos) |
| Random Search | Budget-limited | Often good | Large search space |
| Bayesian Optimization | Adaptive | Best with budget | Expensive training runs |
| Manual Tuning | Expert-guided | Variable | Initial exploration |
| Population-Based | Parallel adaptive | Excellent | Access to multiple GPUs |

*Figure: Hyperparameter Search Strategies*

# Part III Capstone Lab

## 🔬 LAB: Complete Data Pipeline: From Raw Data to Model-Ready Dataset

This lab walks you through building a complete, production-quality data pipeline that loads raw data, performs exploratory analysis, preprocesses and tokenizes text, creates proper train/validation/test splits, and outputs a model-ready dataset saved to Google Drive.

**Objective:** Process the AG News dataset through a complete pipeline including profiling, cleaning, tokenization, and collation. Save the processed dataset to Drive for use in Part IV training labs.

**Duration:** 60–90 minutes

```
# Part III Capstone: Complete Data Pipeline
from datasets import load_dataset, DatasetDict
from transformers import AutoTokenizer, DataCollatorWithPadding
import numpy as np
from collections import Counter

# ============================================
# STEP 1: Load and Explore
# ============================================
print('STEP 1: Loading dataset...')
raw_dataset = load_dataset('ag_news')
print(f'  Splits: {list(raw_dataset.keys())}')
print(f'  Train: {len(raw_dataset["train"]):,}')
print(f'  Test: {len(raw_dataset["test"]):,}')
print(f'  Features: {raw_dataset["train"].features}')
print(f'  Labels: {raw_dataset["train"].features["label"].names}')

# ============================================
# STEP 2: Exploratory Data Analysis
# ============================================
print('\nSTEP 2: Analyzing data...')
class_names = raw_dataset['train'].features['label'].names

# Class distribution
dist = Counter(raw_dataset['train']['label'])
print('  Class distribution:')
for cls_id, count in sorted(dist.items()):
    name = class_names[cls_id]
    pct = count / len(raw_dataset['train']) * 100
```

```
    print(f'    {name:>12}: {count:>6,} ({pct:.1f}%)')

# Text length statistics
lengths = [len(t.split()) for t in raw_dataset['train']['text']]
print(f'  Text lengths (words):')
print(f'    Min={min(lengths)}, Max={max(lengths)},
Mean={np.mean(lengths):.0f}')
print(f'    Median={np.median(lengths):.0f},
95th={np.percentile(lengths,95):.0f}')

# ================================================
# STEP 3: Create validation split
# ================================================
print('\nSTEP 3: Creating train/val/test splits...')
train_val = raw_dataset['train'].train_test_split(test_size=0.1, seed=42,
    stratify_by_column='label')
dataset = DatasetDict({
    'train': train_val['train'],
    'validation': train_val['test'],
    'test': raw_dataset['test'],
})
for split, ds in dataset.items():
    print(f'  {split:>12}: {len(ds):>6,} examples')

# ================================================
# STEP 4: Tokenize
# ================================================
print('\nSTEP 4: Tokenizing...')
model_name = 'distilbert-base-uncased'
tokenizer = AutoTokenizer.from_pretrained(model_name)
MAX_LENGTH = 128  # Based on 95th percentile analysis

def tokenize_fn(batch):
    return tokenizer(batch['text'], truncation=True,
max_length=MAX_LENGTH)

tokenized = dataset.map(tokenize_fn, batched=True, batch_size=1000,
    num_proc=2, desc='Tokenizing')

# ================================================
# STEP 5: Clean up and format
```

```
# ============================================
print('\nSTEP 5: Cleaning up...')
tokenized = tokenized.remove_columns(['text'])
tokenized = tokenized.rename_column('label', 'labels')
tokenized.set_format('torch')

print(f'  Final columns: {tokenized["train"].column_names}')
print(f'  Sample: {tokenized["train"][0]}')

# ============================================
# STEP 6: Save to Drive
# ============================================
print('\nSTEP 6: Saving to Drive...')
save_path = '/content/drive/MyDrive/hf_research/data/ag_news_processed'
tokenized.save_to_disk(save_path)
print(f'  Saved to: {save_path}')

# Verify
from datasets import load_from_disk
loaded = load_from_disk(save_path)
print(f'  Verified: {loaded}')
print('\nPipeline complete! Data ready for training.')
```

| Pipeline Stage | Output | Time Taken | Notes |
|---|---|---|---|
| 1. Load | Raw DatasetDict | (Record) | |
| 2. EDA | Statistics and distributions | (Record) | |
| 3. Split | Train/Val/Test DatasetDict | (Record) | |
| 4. Tokenize | Tokenized dataset | (Record) | |
| 5. Format | Torch-formatted dataset | (Record) | |
| 6. Save | Dataset on Google Drive | (Record) | |

*Figure: Part III Capstone Pipeline Execution Log*

# PART IV

## TRAINING AND FINE-TUNING

# Chapter 12. Fine-Tuning Transformer Models in Google Colab

Fine-tuning is the process of adapting a pre-trained model to a specific task by training it on task-specific data. This chapter provides a comprehensive guide to fine-tuning transformer models using the Hugging Face Trainer API within the constraints of Google Colab.

## 12.1 The Trainer API

The Trainer API is the recommended way to fine-tune models in the Hugging Face ecosystem. It handles the training loop, evaluation, logging, checkpointing, gradient accumulation, mixed precision, and distributed training in a unified interface that has been extensively tested and optimized.

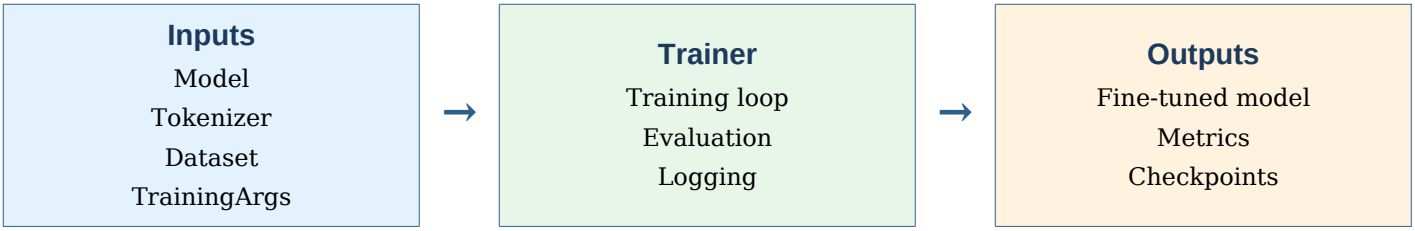


*Figure: Trainer API component overview*

```
from transformers import (AutoModelForSequenceClassification,
AutoTokenizer,
                          Trainer, TrainingArguments,
DataCollatorWithPadding)
from datasets import load_dataset
import evaluate, numpy as np

# 1. Load model and tokenizer
model_name = 'distilbert-base-uncased'
tokenizer = AutoTokenizer.from_pretrained(model_name)
model = AutoModelForSequenceClassification.from_pretrained(model_name,
num_labels=2)

# 2. Prepare dataset
dataset = load_dataset('imdb')
def tokenize(batch):
    return tokenizer(batch['text'], truncation=True, max_length=256)
```

```
tokenized = dataset.map(tokenize, batched=True)
tokenized = tokenized.remove_columns(['text'])
tokenized.set_format('torch')

# 3. Define training arguments
training_args = TrainingArguments(
    output_dir='./results',
    num_train_epochs=3,
    per_device_train_batch_size=16,
    per_device_eval_batch_size=64,
    learning_rate=2e-5,
    weight_decay=0.01,
    evaluation_strategy='epoch',
    save_strategy='epoch',
    load_best_model_at_end=True,
    logging_steps=100,
    fp16=True,  # Mixed precision for speed
)

# 4. Define metrics
accuracy = evaluate.load('accuracy')
def compute_metrics(eval_pred):
    logits, labels = eval_pred
    preds = np.argmax(logits, axis=-1)
    return accuracy.compute(predictions=preds, references=labels)

# 5. Create Trainer and train
trainer = Trainer(
    model=model,
    args=training_args,
    train_dataset=tokenized['train'],
    eval_dataset=tokenized['test'],
    tokenizer=tokenizer,
    data_collator=DataCollatorWithPadding(tokenizer),
    compute_metrics=compute_metrics,
)

trainer.train()
```

## 12.2 Key TrainingArguments

| Parameter | Default | Recommended | Description |
|---|---|---|---|
| learning_rate | 5e-5 | 2e-5 | Start lower for fine-tuning |
| num_train_epochs | 3 | 3–5 | More for small datasets |
| per_device_train_batch_size | 8 | 16–32 | Larger if memory allows |
| weight_decay | 0 | 0.01 | Regularization |
| warmup_ratio | 0 | 0.06–0.1 | Gradual LR warmup |
| fp16 | False | True | 2x speed, half memory |
| gradient_accumulation_steps | 1 | 2–8 | Simulates larger batch |
| save_total_limit | None | 2–3 | Limit disk usage |
| evaluation_strategy | 'no' | 'epoch' | Track overfitting |
| load_best_model_at_end | False | True | Auto-select best checkpoint |

*Figure: Key TrainingArguments for Fine-Tuning in Colab*

## 12.3 Evaluation During Training

Monitoring metrics during training helps detect overfitting early. The Trainer computes evaluation metrics at intervals specified by evaluation_strategy. Combined with load_best_model_at_end, this ensures you always end up with the best model.

## 12.4 Checkpointing and Recovery

```
# Save to Google Drive for persistence
training_args = TrainingArguments(
    output_dir='/content/drive/MyDrive/hf_research/checkpoints',
    save_strategy='epoch',
    save_total_limit=3,
)

# Resume from checkpoint after Colab disconnect
trainer.train(resume_from_checkpoint=True)
```

> ⚠️ **Warning:** *Always point output_dir to Google Drive when training in Colab. Temporary storage is lost when the session ends.*

## 🔬 LAB: Fine-Tune DistilBERT for Sentiment Analysis

Complete the full fine-tuning pipeline above with the IMDB dataset. Track training loss and evaluation accuracy across epochs. Save the final model to Google Drive.

| Epoch | Train Loss | Eval Loss | Eval Accuracy | Time (min) |
|---|---|---|---|---|
| 1 | (Record) | (Record) | (Record) | (Record) |
| 2 | (Record) | (Record) | (Record) | (Record) |
| 3 | (Record) | (Record) | (Record) | (Record) |

*Figure: Lab 12 Training Log*

## Complete Fine-Tuning Workflow: Step by Step

Let us walk through a complete fine-tuning workflow in detail, explaining the purpose and rationale behind each step. This extended example fine-tunes DistilBERT on the AG News dataset for topic classification, demonstrating all the important concepts covered in this chapter.

```
# COMPLETE FINE-TUNING WORKFLOW
# Step 1: Install and import
!pip install -q transformers datasets evaluate accelerate

import torch
import numpy as np
from transformers import (AutoModelForSequenceClassification,
    AutoTokenizer, Trainer, TrainingArguments,
    DataCollatorWithPadding, EarlyStoppingCallback)
from datasets import load_dataset
import evaluate

# Step 2: Set seeds for reproducibility
from transformers import set_seed
SEED = 42
set_seed(SEED)

# Step 3: Load dataset
dataset = load_dataset('ag_news')
print(f'Classes: {dataset["train"].features["label"].names}')
print(f'Train: {len(dataset["train"]):,} examples')
print(f'Test: {len(dataset["test"]):,} examples')

# Step 4: Create validation split
split = dataset['train'].train_test_split(test_size=0.1, seed=SEED)
train_data = split['train']
val_data = split['test']
test_data = dataset['test']

# Step 5: Load tokenizer and tokenize
model_name = 'distilbert-base-uncased'
tokenizer = AutoTokenizer.from_pretrained(model_name)

def tokenize_fn(batch):
```

```
    return tokenizer(batch['text'], truncation=True, max_length=256)

train_tok = train_data.map(tokenize_fn, batched=True, batch_size=1000)
val_tok = val_data.map(tokenize_fn, batched=True, batch_size=1000)
test_tok = test_data.map(tokenize_fn, batched=True, batch_size=1000)

# Remove raw text column, set format
for ds in [train_tok, val_tok, test_tok]:
    ds.set_format('torch', columns=['input_ids', 'attention_mask',
'label'])

# Step 6: Load model
model = AutoModelForSequenceClassification.from_pretrained(
    model_name, num_labels=4)
print(f'Model parameters: {sum(p.numel() for p in model.parameters()):,}')

# Step 7: Define metrics
accuracy = evaluate.load('accuracy')
f1 = evaluate.load('f1')

def compute_metrics(eval_pred):
    logits, labels = eval_pred
    preds = np.argmax(logits, axis=-1)
    acc = accuracy.compute(predictions=preds, references=labels)
    f1_score = f1.compute(predictions=preds, references=labels,
average='macro')
    return {'accuracy': acc['accuracy'], 'f1': f1_score['f1']}

# Step 8: Configure training
training_args = TrainingArguments(
    output_dir='/content/drive/MyDrive/hf_research/ag_news_ft',
    num_train_epochs=5,
    per_device_train_batch_size=32,
    per_device_eval_batch_size=64,
    learning_rate=3e-5,
    weight_decay=0.01,
    warmup_ratio=0.1,
    evaluation_strategy='epoch',
    save_strategy='epoch',
    load_best_model_at_end=True,
    metric_for_best_model='f1',
```

```
    greater_is_better=True,
    fp16=True,
    logging_steps=50,
    save_total_limit=2,
    report_to='none',
    seed=SEED,
)

# Step 9: Create Trainer
trainer = Trainer(
    model=model,
    args=training_args,
    train_dataset=train_tok,
    eval_dataset=val_tok,
    tokenizer=tokenizer,
    data_collator=DataCollatorWithPadding(tokenizer),
    compute_metrics=compute_metrics,
    callbacks=[EarlyStoppingCallback(early_stopping_patience=2)],
)

# Step 10: Train
train_result = trainer.train()
print(f'\nTraining completed!')
print(f'Total steps: {train_result.global_step}')
print(f'Training loss: {train_result.training_loss:.4f}')

# Step 11: Evaluate on test set
test_results = trainer.evaluate(test_tok)
print(f'\nTest Results:')
for key, value in test_results.items():
    if 'eval_' in key:
        print(f'  {key}: {value:.4f}')

# Step 12: Save model and config
save_dir = '/content/drive/MyDrive/hf_research/ag_news_ft/final'
trainer.save_model(save_dir)
tokenizer.save_pretrained(save_dir)

import json
with open(f'{save_dir}/experiment_config.json', 'w') as f:
```

```
    json.dump({'model': model_name, 'lr': 3e-5, 'epochs': 5, 'seed':
SEED}, f)
print(f'Model saved to: {save_dir}')
```

### Understanding the Training Loop

While the Trainer API abstracts away the training loop, understanding what happens behind the scenes helps you debug issues and make informed decisions about hyperparameters. Here is a simplified version of what the Trainer does in each step:

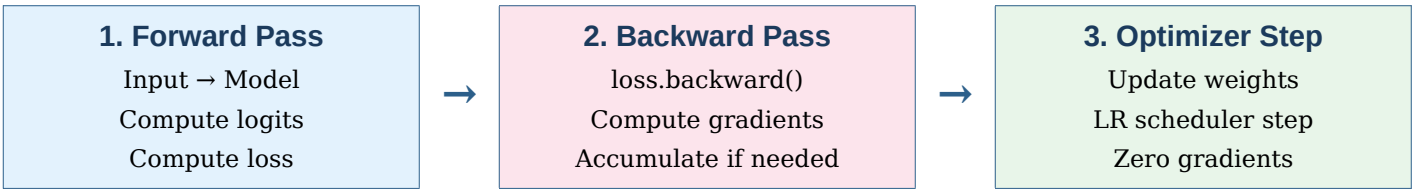


*Figure: The three phases of each training step*

| Hyperparameter | Too Low | Too High | How to Tell |
|---|---|---|---|
| Learning Rate | Training too slow, underfitting | Divergence, NaN loss | Loss should decrease steadily |
| Batch Size | Noisy gradients, slow convergence | OOM error, poor generalization | Use largest that fits in GPU |
| Epochs | Underfitting, low accuracy | Overfitting (val loss increases) | Watch eval metrics each epoch |
| Weight Decay | Overfitting | Underfitting | Usually 0.01 works well |
| Warmup Ratio | Unstable early training | Wastes training time | 5-10% of total steps |
| Max Length | Loses information | Wastes memory | Analyze text length distribution |

*Figure: Hyperparameter Tuning Guide: Symptoms and Diagnosis*

# Chapter 13. Parameter-Efficient Fine-Tuning (PEFT)

Parameter-efficient fine-tuning techniques allow you to adapt large models using a fraction of the parameters and memory required for full fine-tuning.

This is essential for working with large models in Colab's constrained environment.

## 13.1 LoRA: Low-Rank Adaptation

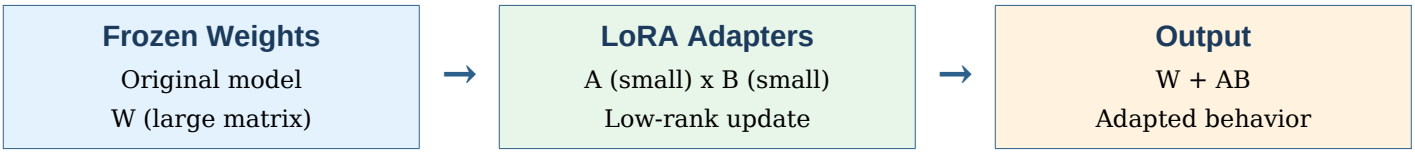


*Figure: LoRA: Adding low-rank adapter matrices to frozen model weights*

| Method | Trainable Params | Memory Saving | Quality | Speed |
|---|---|---|---|---|
| Full Fine-Tuning | 100% | None | Best | Slowest |
| LoRA (r=16) | 0.5–2% | 60–80% | Near-best | Fast |
| QLoRA (4-bit + LoRA) | 0.5–2% | 85–95% | Good | Medium |
| Prefix Tuning | 0.1–1% | 70–85% | Good | Fast |
| Adapters (Houlsby) | 3–5% | 50–70% | Very good | Medium |

*Figure: Comparison of Parameter-Efficient Fine-Tuning Methods*

```
from peft import LoraConfig, get_peft_model, TaskType
from transformers import AutoModelForSequenceClassification

# Load base model
model = AutoModelForSequenceClassification.from_pretrained(
    'bert-base-uncased', num_labels=2)

# Configure LoRA
lora_config = LoraConfig(
    task_type=TaskType.SEQ_CLS,
    r=16,               # Rank of decomposition
    lora_alpha=32,      # Scaling factor
    lora_dropout=0.1,   # Dropout on adapter
    target_modules=['query', 'value'],  # Which layers to adapt
```

```
)

# Apply LoRA
model = get_peft_model(model, lora_config)
model.print_trainable_parameters()
# trainable params: 887,042 || all params: 109,895,940 || trainable%: 0.81
```

## 13.2 QLoRA: Quantized LoRA

```
from transformers import AutoModelForCausalLM, BitsAndBytesConfig
import torch

# Load model in 4-bit
bnb_config = BitsAndBytesConfig(
    load_in_4bit=True,
    bnb_4bit_compute_dtype=torch.float16,
    bnb_4bit_quant_type='nf4',
    bnb_4bit_use_double_quant=True,
)

model = AutoModelForCausalLM.from_pretrained(
    'meta-llama/Llama-2-7b-hf',
    quantization_config=bnb_config,
    device_map='auto',
)

# Then apply LoRA on top
from peft import LoraConfig, get_peft_model
lora_config = LoraConfig(r=16, lora_alpha=32,
    target_modules=['q_proj', 'v_proj', 'k_proj', 'o_proj'],
    lora_dropout=0.05, bias='none')
model = get_peft_model(model, lora_config)
```

## 13.3 Choosing the Right PEFT Method

| Scenario | Recommended Method | Why |
|---|---|---|
| < 200M params, enough VRAM | Full fine-tuning | Best quality, fast enough |
| 1-3B params, 16GB | LoRA (r=16-32) | Good balance of quality and speed |

| Scenario | Recommended Method | Why |
|---|---|---|
| VRAM | | |
| 7B+ params, 16GB VRAM | QLoRA (4-bit + LoRA) | Only way to fit in memory |
| Multiple tasks, one model | LoRA adapters | Small, swappable adapters |
| Minimal code changes | LoRA via PEFT library | Drop-in compatible with Trainer |

*Figure: PEFT Method Selection Guide*

## 🔬 LAB: LoRA Fine-Tuning of BERT for Text Classification

Fine-tune BERT using LoRA on the AG News dataset. Compare training speed, memory usage, and final accuracy against full fine-tuning from Chapter 12.

## Understanding LoRA Mathematics

To use LoRA effectively, it helps to understand the mathematics behind it. In a standard transformer layer, the attention mechanism uses weight matrices W_q, W_k, W_v, and W_o to transform the input into queries, keys, values, and output projections. Each of these matrices has dimensions d x d, where d is the model's hidden dimension (e.g., 768 for BERT-base, 4096 for LLaMA-7B).

During full fine-tuning, each of these matrices receives gradient updates that change all d x d parameters. LoRA observes that these updates tend to have low rank, meaning that the update matrix delta_W can be approximated by the product of two much smaller matrices: A (d x r) and B (r x d), where r is much smaller than d (typically 8-64 while d is 768-4096).

The key equation is: W_new = W_original + A * B, where W_original is frozen and only A and B are trained. This reduces the number of trainable parameters from d^2 to 2 * d * r. For example, with d=4096 and r=16, you go from 16.7 million parameters per matrix to just 131,072, which is a 99.2% reduction.

| Model Size | Hidden Dim (d) | Full FT Params/Layer | LoRA (r=16) | Reduction |
|---|---|---|---|---|
| BERT-base | 768 | 590K | 24.6K | 95.8% |
| BERT-large | 1024 | 1.05M | 32.8K | 96.9% |
| LLaMA-7B | 4096 | 16.8M | 131K | 99.2% |
| LLaMA-13B | 5120 | 26.2M | 164K | 99.4% |
| LLaMA-70B | 8192 | 67.1M | 262K | 99.6% |

*Figure: Parameter Reduction with LoRA (r=16) by Model Size*

## Choosing LoRA Hyperparameters

The three most important LoRA hyperparameters are the rank (r), the scaling factor (alpha), and the target modules. Each affects the fine-tuning result in different ways.

| Hyperparameter | Range | Effect of Increasing | Recommendation |
|---|---|---|---|
| r (rank) | 4–64 | More capacity, more memory | Start with 16, increase if underfitting |
| lora_alpha | 16–64 | Larger effective learning rate | Usually 2x rank (e.g., r=16, alpha=32) |

| Hyperparameter | Range | Effect of Increasing | Recommendation |
|---|---|---|---|
| target_modules | q,v to all | More adaptation points | Start with q,v; add k,o if needed |
| lora_dropout | 0.0–0.1 | More regularization | 0.05–0.1 for small datasets |
| bias | none/all | Trains bias terms too | 'none' is usually sufficient |

*Figure: LoRA Hyperparameter Guide*

## Comparing PEFT Methods on Real Tasks

```
# Benchmark LoRA vs Full Fine-Tuning
import time, torch
from transformers import AutoModelForSequenceClassification,
AutoTokenizer, Trainer, TrainingArguments
from peft import LoraConfig, get_peft_model, TaskType
from datasets import load_dataset

model_name = 'distilbert-base-uncased'
dataset = load_dataset('imdb', split='train[:2000]')
tokenizer = AutoTokenizer.from_pretrained(model_name)

def tokenize(batch):
    return tokenizer(batch['text'], truncation=True, max_length=256)
dataset = dataset.map(tokenize, batched=True).remove_columns(['text'])
dataset.set_format('torch')

results = {}

# Full fine-tuning
model_full =
AutoModelForSequenceClassification.from_pretrained(model_name,
num_labels=2)
full_params = sum(p.numel() for p in model_full.parameters() if
p.requires_grad)
torch.cuda.reset_peak_memory_stats()
start = time.time()
# ... train for 1 epoch ...
results['Full FT'] = {
    'params': f'{full_params:,}',
    'peak_mem': f'{torch.cuda.max_memory_allocated()/1e9:.2f} GB',
}
```

```
# LoRA fine-tuning
model_lora =
AutoModelForSequenceClassification.from_pretrained(model_name,
num_labels=2)
lora_config = LoraConfig(task_type=TaskType.SEQ_CLS, r=16, lora_alpha=32)
model_lora = get_peft_model(model_lora, lora_config)
lora_params = sum(p.numel() for p in model_lora.parameters() if
p.requires_grad)
results['LoRA'] = {
    'params': f'{lora_params:,}',
}

print('Comparison:')
for method, r in results.items():
    print(f'  {method}: {r}')
```

# Chapter 14. Working with Large Models Under Limited Resources

## 14.1 Quantization

| Precision | Bits/Param | Memory (7B model) | Quality Impact | Speed Impact |
|---|---|---|---|---|
| FP32 | 32 bits | 28 GB | Baseline | Baseline |
| FP16 / BF16 | 16 bits | 14 GB | Negligible | 2x faster |
| INT8 | 8 bits | 7 GB | Minimal | 1.5–2x |
| NF4 (4-bit) | 4 bits | 3.5 GB | Small | Variable |
| GPTQ (4-bit) | 4 bits | 3.5 GB | Small | Faster inference |
| AWQ (4-bit) | 4 bits | 3.5 GB | Minimal | Fastest inference |

*Figure: Quantization Methods and Their Impact*

## 14.2 Mixed Precision Training

```
# Enable mixed precision in TrainingArguments
training_args = TrainingArguments(
```

```
    fp16=True,   # For NVIDIA T4, V100
    # bf16=True, # For A100 (better numerical stability)
    ...
)
```

## 14.3 Device Mapping and Model Parallelism

```
# Automatic device mapping
model = AutoModelForCausalLM.from_pretrained(
    'bigscience/bloom-3b',
    device_map='auto',        # Spread across GPU + CPU
    torch_dtype=torch.float16,
)
print(model.hf_device_map)  # Shows layer-to-device mapping
```

## 14.4 Memory Optimization Techniques

| Technique | Memory Saving | Speed Impact | Code Change |
|---|---|---|---|
| Reduce batch size | Linear | Slower training | batch_size=8 to 4 |
| Gradient accumulation | None (same effective) | Slightly slower | gradient_accumulation_steps=4 |
| Gradient checkpointing | 50–70% | 20–30% slower | gradient_checkpointing=True |
| Mixed precision (FP16) | ~50% | Faster | fp16=True |
| 4-bit quantization | 75–87% | Variable | BitsAndBytesConfig |
| Flash Attention 2 | 40–60% | Faster | attn_implementation='flash_attention_2' |
| torch.cuda.empty_cache() | Frees unused | None | Call between experiments |

*Figure: Memory Optimization Techniques for Colab*

### 🔬 LAB: Running a 7B Parameter Model in Colab Free Tier

Load and run inference with a 7B parameter model using 4-bit quantization. Measure memory usage and inference speed.

## Quantization in Practice: A Complete Example

Let us walk through a complete example of loading, quantizing, and using a large language model in Colab. We will use the bitsandbytes library to load a model in 4-bit precision and demonstrate both inference and fine-tuning.

```
# Complete 4-bit quantization example
!pip install -q bitsandbytes accelerate

import torch
from transformers import (
    AutoModelForCausalLM, AutoTokenizer,
    BitsAndBytesConfig, pipeline
)

# Step 1: Configure 4-bit quantization
bnb_config = BitsAndBytesConfig(
    load_in_4bit=True,
    bnb_4bit_compute_dtype=torch.float16,
    bnb_4bit_quant_type='nf4',       # NormalFloat4 quantization
    bnb_4bit_use_double_quant=True,  # Nested quantization
)

# Step 2: Load model with quantization
model_name = 'mistralai/Mistral-7B-Instruct-v0.2'
print(f'Loading {model_name} in 4-bit...')

tokenizer = AutoTokenizer.from_pretrained(model_name)
model = AutoModelForCausalLM.from_pretrained(
    model_name,
    quantization_config=bnb_config,
    device_map='auto',
)

# Step 3: Check memory usage
mem_alloc = torch.cuda.memory_allocated() / 1e9
print(f'GPU memory used: {mem_alloc:.2f} GB')
print(f'(Full FP16 would need ~14 GB)')
print(f'(4-bit uses only ~{mem_alloc:.1f} GB = {mem_alloc/14*100:.0f}% of
FP16)')
```

```
# Step 4: Run inference
messages = [{'role': 'user', 'content': 'Explain quantum computing in 3
sentences.'}]
input_text = tokenizer.apply_chat_template(messages, tokenize=False)
inputs = tokenizer(input_text, return_tensors='pt').to(model.device)

with torch.no_grad():
    outputs = model.generate(**inputs, max_new_tokens=150,
                             temperature=0.7, do_sample=True)

response = tokenizer.decode(outputs[0], skip_special_tokens=True)
print(f'\nResponse: {response}')

# Step 5: Quick benchmark
import time
start = time.time()
for _ in range(5):
    with torch.no_grad():
        _ = model.generate(**inputs, max_new_tokens=50)
avg_time = (time.time() - start) / 5
tokens_per_sec = 50 / avg_time
print(f'\nAverage generation: {avg_time:.2f}s for 50 tokens')
print(f'Speed: {tokens_per_sec:.1f} tokens/second')
```

| Quantization Method | Library | Compression | Inference Speed | Quality |
|---|---|---|---|---|
| bitsandbytes NF4 | bitsandbytes | 8x (4-bit) | Moderate | Very good |
| bitsandbytes INT8 | bitsandbytes | 4x (8-bit) | Good | Excellent |
| GPTQ | auto-gptq | 8x (4-bit) | Fast (GPU-optimized) | Very good |
| AWQ | autoawq | 8x (4-bit) | Fastest | Good |
| GGUF | llama.cpp | Variable (2-8 bit) | Good (CPU too) | Varies |
| EETQ | eetq | 4x (8-bit) | Very fast | Good |

*Figure: Quantization Methods Comparison*

### Flash Attention and Memory-Efficient Attention

Flash Attention is an algorithm that computes exact attention with significantly less memory usage and improved speed. It achieves this by tiling the attention computation to avoid materializing the full attention matrix in GPU memory. For long sequences, this can reduce memory usage from O(n^2) to O(n), enabling processing of much longer inputs.

```
# Using Flash Attention 2 with Hugging Face models
from transformers import AutoModelForCausalLM
import torch

# Load with Flash Attention 2 (requires compatible GPU)
model = AutoModelForCausalLM.from_pretrained(
    'mistralai/Mistral-7B-Instruct-v0.2',
    torch_dtype=torch.float16,
    attn_implementation='flash_attention_2',  # Enable FA2
    device_map='auto',
)

# For training with Flash Attention
from transformers import TrainingArguments
args = TrainingArguments(
    output_dir='./results',
    bf16=True,  # Flash Attention works best with BF16
    # Flash Attention is used automatically if model supports it
)
```

### 🔬 LAB: Large Model Inference Benchmark

Load the same model at different quantization levels and compare memory usage, inference speed, and output quality. Record your results in the table below.

| Precision | GPU Memory | Tokens/ Second | Output Quality (1-5) | Notes |
|---|---|---|---|---|
| FP16 (if fits) | (Record) | (Record) | (Rate) | |
| INT8 | (Record) | (Record) | (Rate) | |
| NF4 (4-bit) | (Record) | (Record) | (Rate) | |
| NF4 + Double | (Record) | (Record) | (Rate) | |

| Precision | GPU Memory | Tokens/ Second | Output Quality (1-5) | Notes |
|---|---|---|---|---|
| Quant | | | | |

*Figure: Lab Results: Quantization Benchmark*

# Chapter 15. Evaluating Model Performance

## 15.1 Standard Metrics by Task

| Task | Primary Metrics | Secondary Metrics | Library |
|---|---|---|---|
| Binary Classification | Accuracy, F1 | AUC-ROC, Precision, Recall | evaluate |
| Multi-class Classification | Macro F1, Accuracy | Per-class F1, Confusion Matrix | evaluate + sklearn |
| NER | Entity-level F1 | Precision, Recall by entity type | seqeval |
| Summarization | ROUGE-1, ROUGE-2, ROUGE-L | BERTScore, factual consistency | evaluate |
| Translation | BLEU, chrF | TER, COMET | evaluate + sacrebleu |
| Question Answering | Exact Match, F1 | Has-answer accuracy | evaluate |
| Language Modeling | Perplexity | Bits per character | evaluate |

*Figure: Evaluation Metrics by Task Type*

```
import evaluate
import numpy as np

# Combine multiple metrics
clf_metrics = evaluate.combine(['accuracy', 'f1', 'precision', 'recall'])

# Simulate predictions
predictions = [1, 0, 1, 1, 0, 1, 0, 0, 1, 1]
references  = [1, 0, 0, 1, 0, 1, 1, 0, 1, 0]

results = clf_metrics.compute(predictions=predictions,
references=references)
print('Evaluation Results:')
for metric, value in results.items():
    print(f'  {metric:>12}: {value:.4f}')
```

## 15.2 Error Analysis Workflow

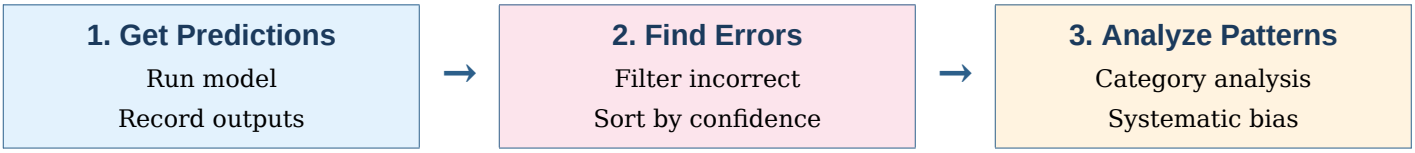


*Figure: Error analysis workflow*

## 15.3 Practical Evaluation Habits

| Habit | Why It Matters |
|---|---|
| Always use held-out test set | Prevents overfitting to evaluation data |
| Report mean and std over 3+ runs | Accounts for random variation |
| Use same preprocessing for all models | Ensures fair comparison |
| Include baseline models | Provides context for improvements |
| Check per-class performance | Reveals hidden weaknesses |
| Save predictions, not just metrics | Enables later analysis |

*Figure: Evaluation Best Practices*

### 🔬 LAB: Comprehensive Model Evaluation

Evaluate a fine-tuned model using multiple metrics, perform error analysis, and create a detailed evaluation report.

## Building an Evaluation Report

A comprehensive evaluation report goes beyond aggregate metrics. It should include performance breakdowns by category, confidence distribution analysis, example-level inspection, and comparison with baselines. Here is a template for building such a report.

```
import numpy as np
from collections import defaultdict

def comprehensive_evaluation(predictions, references, texts, class_names):
    """Generate a comprehensive evaluation report."""
    import evaluate

    # Overall metrics
    accuracy = evaluate.load('accuracy')
    f1 = evaluate.load('f1')

    overall = {
        'accuracy': accuracy.compute(predictions=predictions,
references=references)['accuracy'],
        'macro_f1': f1.compute(predictions=predictions,
references=references, average='macro')['f1'],
    }

    # Per-class metrics
    per_class = {}
    for cls_id, cls_name in enumerate(class_names):
        mask = [r == cls_id for r in references]
        cls_preds = [p for p, m in zip(predictions, mask) if m]
        cls_refs = [r for r, m in zip(references, mask) if m]
        if cls_preds:
            per_class[cls_name] = {
                'count': len(cls_preds),
                'accuracy': sum(p == r for p, r in zip(cls_preds,
cls_refs)) / len(cls_preds),
            }

    # Error analysis
    errors = []
    for i, (pred, ref, text) in enumerate(zip(predictions, references,
texts)):
```

```
        if pred != ref:
            errors.append({
                'index': i,
                'predicted': class_names[pred],
                'actual': class_names[ref],
                'text': text[:100],
            })

    # Print report
    print('=' * 60)
    print('EVALUATION REPORT')
    print('=' * 60)
    print(f'\nOverall Accuracy: {overall["accuracy"]:.4f}')
    print(f'Macro F1 Score:   {overall["macro_f1"]:.4f}')
    print(f'\nPer-Class Performance:')
    for cls, metrics in per_class.items():
        bar = '#' * int(metrics['accuracy'] * 30)
        print(f'  {cls:>15}: {bar} {metrics["accuracy"]:.4f}
(n={metrics["count"]})')
    print(f'\nTotal Errors: {len(errors)} / {len(predictions)}
({len(errors)/len(predictions)*100:.1f}%)')
    print(f'\nSample Errors (first 5):')
    for err in errors[:5]:
        print(f'  Predicted: {err["predicted"]:>10} | Actual:
{err["actual"]:>10}')
        print(f'  Text: "{err["text"]}..."')
        print()

    return overall, per_class, errors
```

## Statistical Significance in Model Comparison

When comparing two models, it is not enough to show that one has a higher accuracy than the other. The difference might be due to random variation rather than a genuine performance difference. Statistical significance testing helps you determine whether the observed difference is likely to be real.

| Test | When to Use | Requirements | Interpretation |
|---|---|---|---|
| Paired t-test | Comparing two models on same data | Normally distributed differences | $p < 0.05$ = significant |

| Test | When to Use | Requirements | Interpretation |
|---|---|---|---|
| Wilcoxon signed-rank | Non-parametric alternative | Paired observations | $p < 0.05$ = significant |
| McNemar's test | Comparing classification accuracy | Binary correct/incorrect | $p < 0.05$ = significant |
| Bootstrap confidence interval | Any metric comparison | Large enough sample | No CI overlap = significant |

*Figure: Statistical Tests for Model Comparison*

```
# Bootstrap confidence interval for model comparison
import numpy as np

def bootstrap_ci(metric_fn, predictions, references, n_bootstrap=1000,
ci=0.95):
    """Compute bootstrap confidence interval for a metric."""
    scores = []
    n = len(predictions)
    for _ in range(n_bootstrap):
        idx = np.random.choice(n, size=n, replace=True)
        boot_preds = [predictions[i] for i in idx]
        boot_refs = [references[i] for i in idx]
        score = sum(p == r for p, r in zip(boot_preds, boot_refs)) / n
        scores.append(score)

    lower = np.percentile(scores, (1-ci)/2 * 100)
    upper = np.percentile(scores, (1+ci)/2 * 100)
    mean = np.mean(scores)
    return mean, lower, upper

# Example usage
mean, lo, hi = bootstrap_ci(None, predictions=[1,0,1,1,0]*100,
references=[1,0,1,0,0]*100)
print(f'Accuracy: {mean:.4f} (95% CI: [{lo:.4f}, {hi:.4f}])')
```

# Part IV Capstone Lab

## 🔬 LAB: Full Fine-Tuning vs LoRA: A Controlled Comparison

In this capstone lab, you will fine-tune the same base model on the same dataset using both full fine-tuning and LoRA, then rigorously compare the results. This head-to-head comparison will give you practical intuition for when to use each approach.

**Objective:** Train DistilBERT on AG News using (a) full fine-tuning and (b) LoRA fine-tuning. Compare training time, memory usage, final accuracy, and per-class performance.

**Duration:** 120–150 minutes (including training time)

```
# Part IV Capstone: Full FT vs LoRA Comparison
import torch, time, json
import numpy as np
from transformers import (AutoModelForSequenceClassification,
AutoTokenizer,
    Trainer, TrainingArguments, DataCollatorWithPadding, set_seed)
from datasets import load_from_disk
import evaluate

set_seed(42)
MODEL_NAME = 'distilbert-base-uncased'

# Load preprocessed dataset from Part III lab
dataset =
load_from_disk('/content/drive/MyDrive/hf_research/data/ag_news_processed'
)
# Or create a smaller version for faster experimentation:
# dataset['train'] = dataset['train'].select(range(5000))

tokenizer = AutoTokenizer.from_pretrained(MODEL_NAME)
collator = DataCollatorWithPadding(tokenizer)

accuracy = evaluate.load('accuracy')
f1 = evaluate.load('f1')

def compute_metrics(eval_pred):
    logits, labels = eval_pred
    preds = np.argmax(logits, axis=-1)
    return {
```

```
        'accuracy': accuracy.compute(predictions=preds, references=labels)
['accuracy'],
        'f1_macro': f1.compute(predictions=preds, references=labels,
average='macro')['f1'],
    }

comparison = {}

# ============================================
# EXPERIMENT A: Full Fine-Tuning
# ============================================
print('EXPERIMENT A: Full Fine-Tuning')
print('=' * 50)

model_full = AutoModelForSequenceClassification.from_pretrained(
    MODEL_NAME, num_labels=4)
total_params = sum(p.numel() for p in model_full.parameters())
trainable_params = sum(p.numel() for p in model_full.parameters() if
p.requires_grad)
print(f'Total params: {total_params:,}')
print(f'Trainable params: {trainable_params:,} (100%)')

torch.cuda.reset_peak_memory_stats()
start_time = time.time()

args_full = TrainingArguments(
    output_dir='./full_ft', num_train_epochs=3, learning_rate=2e-5,
    per_device_train_batch_size=32, per_device_eval_batch_size=64,
    evaluation_strategy='epoch', save_strategy='epoch',
    load_best_model_at_end=True, metric_for_best_model='f1_macro',
    fp16=True, logging_steps=100, save_total_limit=1, report_to='none')

trainer_full = Trainer(model=model_full, args=args_full,
    train_dataset=dataset['train'], eval_dataset=dataset['validation'],
    tokenizer=tokenizer, data_collator=collator,
compute_metrics=compute_metrics)

trainer_full.train()
full_time = time.time() - start_time
full_mem = torch.cuda.max_memory_allocated() / 1e9
full_results = trainer_full.evaluate(dataset['test'])
```

```
comparison['Full FT'] = {
    'trainable_params': f'{trainable_params:,}',
    'peak_gpu_memory': f'{full_mem:.2f} GB',
    'training_time': f'{full_time:.0f}s ({full_time/60:.1f}min)',
    'test_accuracy': round(full_results['eval_accuracy'], 4),
    'test_f1': round(full_results['eval_f1_macro'], 4),
}
del model_full, trainer_full
torch.cuda.empty_cache()

# ============================================
# EXPERIMENT B: LoRA Fine-Tuning
# ============================================
print('\nEXPERIMENT B: LoRA Fine-Tuning')
print('=' * 50)

from peft import LoraConfig, get_peft_model, TaskType

model_lora = AutoModelForSequenceClassification.from_pretrained(
    MODEL_NAME, num_labels=4)
lora_config = LoraConfig(
    task_type=TaskType.SEQ_CLS, r=16, lora_alpha=32,
    lora_dropout=0.1, target_modules=['q_lin', 'v_lin'])
model_lora = get_peft_model(model_lora, lora_config)
model_lora.print_trainable_parameters()

lora_trainable = sum(p.numel() for p in model_lora.parameters() if
p.requires_grad)

torch.cuda.reset_peak_memory_stats()
start_time = time.time()

args_lora = TrainingArguments(
    output_dir='./lora_ft', num_train_epochs=3, learning_rate=3e-4,
    per_device_train_batch_size=32, per_device_eval_batch_size=64,
    evaluation_strategy='epoch', save_strategy='epoch',
    load_best_model_at_end=True, metric_for_best_model='f1_macro',
    fp16=True, logging_steps=100, save_total_limit=1, report_to='none')

trainer_lora = Trainer(model=model_lora, args=args_lora,
```

```
    train_dataset=dataset['train'], eval_dataset=dataset['validation'],
    tokenizer=tokenizer, data_collator=collator,
compute_metrics=compute_metrics)

trainer_lora.train()
lora_time = time.time() - start_time
lora_mem = torch.cuda.max_memory_allocated() / 1e9
lora_results = trainer_lora.evaluate(dataset['test'])

comparison['LoRA'] = {
    'trainable_params': f'{lora_trainable:,}',
    'peak_gpu_memory': f'{lora_mem:.2f} GB',
    'training_time': f'{lora_time:.0f}s ({lora_time/60:.1f}min)',
    'test_accuracy': round(lora_results['eval_accuracy'], 4),
    'test_f1': round(lora_results['eval_f1_macro'], 4),
}

# ============================================
# RESULTS COMPARISON
# ============================================
print('\n' + '=' * 60)
print('COMPARISON RESULTS')
print('=' * 60)
for method, metrics in comparison.items():
    print(f'\n{method}:')
    for k, v in metrics.items():
        print(f'  {k:>20}: {v}')
```

| Metric | Full Fine-Tuning | LoRA (r=16) | Difference |
|---|---|---|---|
| Trainable Parameters | (Record) | (Record) | (Calculate) |
| Peak GPU Memory | (Record) | (Record) | (Calculate) |
| Training Time | (Record) | (Record) | (Calculate) |
| Test Accuracy | (Record) | (Record) | (Calculate) |
| Test Macro F1 | (Record) | (Record) | (Calculate) |
| Checkpoint Size (MB) | (Record) | (Record) | (Calculate) |

*Figure: Part IV Capstone Results: Full Fine-Tuning vs LoRA*

| Analysis Question | Your Answer |
|---|---|
| Which method had higher accuracy? By how much? | (Write) |
| Which method used less GPU memory? By what factor? | (Write) |
| Which method trained faster? By what factor? | (Write) |
| Was the accuracy difference statistically significant? | (Write) |
| When would you choose Full FT over LoRA? | (Write) |
| When would you choose LoRA over Full FT? | (Write) |
| What LoRA rank would you try next? Why? | (Write) |

*Figure: Part IV Capstone Analysis Questions*

# PART V

## APPLIED RESEARCH WORKFLOWS

# Chapter 16. Building a Retrieval-Augmented Generation (RAG) Workflow

## 16.1 RAG Architecture Overview

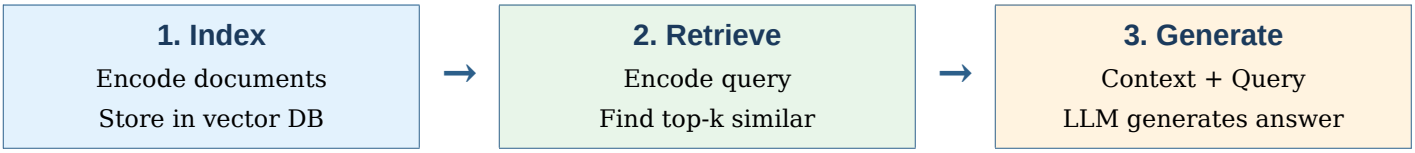


*Figure: RAG Pipeline: Index → Retrieve → Generate*

| RAG Component | Options | Recommended for Colab |
|---|---|---|
| Embedding Model | all-MiniLM-L6-v2, bge-small-en | all-MiniLM-L6-v2 (fast, small) |
| Vector Store | FAISS, ChromaDB, Qdrant | FAISS (simple, efficient) |
| Retrieval | Dense, BM25, Hybrid | Dense retrieval with FAISS |
| Generator | LLaMA, Mistral, Flan-T5 | Flan-T5-base or quantized Mistral |
| Chunking | Fixed-size, semantic, recursive | Recursive with 512 tokens |

*Figure: RAG Component Selection Guide for Colab*

## 16.2 Complete RAG Implementation

```
# Complete RAG prototype in Colab
from sentence_transformers import SentenceTransformer
from transformers import pipeline
import faiss, numpy as np

# Documents (your knowledge base)
documents = [
    'LoRA reduces trainable parameters by 90-99% using low-rank
matrices.',
    'QLoRA combines 4-bit quantization with LoRA for memory efficiency.',
    'The Trainer API handles training loops, evaluation, and
checkpointing.',
    'Flash Attention reduces memory from quadratic to near-linear.',
```

```
    'Gradient checkpointing trades compute for 50-70% memory savings.',
    'Mixed precision training uses FP16 for 2x speed improvement.',
    'FAISS provides efficient approximate nearest-neighbor search.',
    'Hugging Face Hub hosts over 500,000 pre-trained models.',
]

# Step 1: Build index
embedder = SentenceTransformer('all-MiniLM-L6-v2')
doc_embs = embedder.encode(documents, convert_to_numpy=True)
faiss.normalize_L2(doc_embs)
index = faiss.IndexFlatIP(doc_embs.shape[1])
index.add(doc_embs)

# Step 2: Retrieve
query = 'How can I reduce memory usage when training large models?'
q_emb = embedder.encode([query], convert_to_numpy=True)
faiss.normalize_L2(q_emb)
scores, ids = index.search(q_emb, k=3)

context = '\n'.join([documents[i] for i in ids[0]])
print('Retrieved context:', context)

# Step 3: Generate
generator = pipeline('text2text-generation', model='google/flan-t5-base',
device=0)
prompt = f'Based on the following context, answer the question.\n\
nContext: {context}\n\nQuestion: {query}\nAnswer:'
answer = generator(prompt, max_length=200)
print(f'\nAnswer: {answer[0]["generated_text"]}')
```

## 🔬 LAB: Build a Research Paper Q&A System

Extend the RAG prototype with your own research paper abstracts. Build an index of 20+ abstracts and test with 5 different queries.

## RAG Design Patterns

Retrieval-Augmented Generation systems can be designed in several ways depending on the use case. Understanding these design patterns helps you choose the right architecture for your research application.

| Pattern | Architecture | Strengths | Best For |
|---|---|---|---|
| Naive RAG | Retrieve + Generate (simple concat) | Easy to implement | Prototyping, simple Q&A |
| Advanced RAG | Pre-retrieval + Retrieval + Post-retrieval + Generation | Better relevance | Production systems |
| Modular RAG | Interchangeable components | Flexible, testable | Research experiments |
| Multi-hop RAG | Iterative retrieval | Complex reasoning | Multi-step questions |
| Self-RAG | Model decides when to retrieve | Efficient, targeted | Open-domain Q&A |
| Corrective RAG | Retrieval + Verification + Generation | Reduces hallucination | High-stakes applications |

*Figure: RAG Design Patterns*

## Chunking Strategies for RAG

How you split your documents into chunks significantly impacts retrieval quality. The chunking strategy determines the granularity of your retrieval: too large and retrieved chunks contain irrelevant information; too small and they lack sufficient context.

| Strategy | Chunk Size | Overlap | Strengths | Weaknesses |
|---|---|---|---|---|
| Fixed-size | 512 tokens | 50 tokens | Simple, consistent | Breaks mid-sentence |
| Sentence-based | 3-5 sentences | 1 sentence | Natural boundaries | Variable size |
| Paragraph-based | 1 paragraph | None | Semantic coherence | Size varies widely |
| Recursive | 512 tokens (split by \n\n, \n, .) | 50 tokens | Best boundaries | More complex |

| Strategy | Chunk Size | Overlap | Strengths | Weaknesses |
|---|---|---|---|---|
| Semantic | Varies (by similarity) | Adaptive | Optimal coherence | Computationally expensive |

*Figure: Document Chunking Strategies for RAG Systems*

```
# Recursive text splitter implementation
def recursive_split(text, max_size=500, overlap=50):
    """Split text recursively by paragraph, sentence, then character."""
    chunks = []

    # Try splitting by double newline (paragraphs)
    parts = text.split('\n\n')
    current_chunk = ''

    for part in parts:
        if len(current_chunk) + len(part) < max_size:
            current_chunk += part + '\n\n'
        else:
            if current_chunk:
                chunks.append(current_chunk.strip())
            current_chunk = part + '\n\n'

    if current_chunk:
        chunks.append(current_chunk.strip())

    # Add overlap
    overlapped = []
    for i, chunk in enumerate(chunks):
        if i > 0:
            prev_end = chunks[i-1][-overlap:]
            chunk = prev_end + ' ' + chunk
        overlapped.append(chunk)

    return overlapped

# Example usage
chunks = recursive_split(long_document_text)
print(f'Created {len(chunks)} chunks')
for i, chunk in enumerate(chunks[:3]):
```

```
    print(f'  Chunk {i}: {len(chunk)} chars, starts with:
"{chunk[:60]}..."')
```

### Evaluating RAG Systems

| Metric | Measures | How to Compute | Target |
|---|---|---|---|
| Retrieval Precision@k | Relevant docs in top-k | relevant_in_k / k | > 0.7 |
| Retrieval Recall@k | Found relevant docs | found_relevant / total_relevant | > 0.8 |
| Answer Accuracy | Correct final answer | Manual evaluation or EM | > 0.7 |
| Faithfulness | Answer grounded in context | NLI or manual check | > 0.9 |
| Answer Relevancy | Answer addresses question | Semantic similarity | > 0.8 |
| Context Precision | Context items are useful | Used_items / retrieved | > 0.6 |

*Figure: RAG System Evaluation Metrics*

# Chapter 17. Prompt Engineering for Open Models

## 17.1 Prompt Structure

| Component | Purpose | Example |
|---|---|---|
| System instruction | Define role and behavior | "You are a helpful research assistant." |
| Context | Provide relevant information | Retrieved documents, data |
| Few-shot examples | Demonstrate expected format | Input/Output pairs |
| Task instruction | Specify what to do | "Summarize the following text." |

| Component | Purpose | Example |
|---|---|---|
| Output format | Define response structure | "Respond in JSON format." |
| Constraints | Set boundaries | "Use only the provided context." |

*Figure: Prompt Structure Components*

## 17.2 Few-Shot Prompting

```
from transformers import pipeline

generator = pipeline('text-generation', model='gpt2', device=0)

# Zero-shot
prompt_0 = 'Classify the sentiment: "I love this!" -> '

# Few-shot (3 examples)
prompt_3 = '''Classify the sentiment of the text.

Text: "This movie is fantastic!" -> Positive
Text: "Terrible waste of time." -> Negative
Text: "It was okay, nothing special." -> Neutral
Text: "I love this product!" -> '''

for name, prompt in [('Zero-shot', prompt_0), ('Few-shot', prompt_3)]:
    out = generator(prompt, max_new_tokens=5, do_sample=False)
    print(f'{name}: {out[0]["generated_text"]}')
```

## 17.3 Structured Output Generation

## 17.4 Prompt Testing and Iteration

### 🔬 LAB: Prompt Engineering Workshop

Design, test, and iterate on prompts for three different tasks: classification, extraction, and summarization. Record prompt versions and their performance.

## Prompt Templates for Common Research Tasks

Having a library of tested prompt templates saves time and improves consistency. Here are templates for the most common research tasks, ready to use with open models available on the Hugging Face Hub.

| Task | Template Structure | Key Elements |
|---|---|---|
| Summarization | Summarize: {text}\nSummary: | Specify length, focus area |
| Classification | Classify: {text}\nLabel: | List valid labels |
| Extraction | Extract {entity_type} from: {text}\nEntities: | Define entity types clearly |
| Translation | Translate to {lang}: {text}\nTranslation: | Specify target language |
| Q&A | Context: {context}\nQ: {question}\nA: | Separate context from question |
| Code Generation | Write {language} code to: {task}\nCode: | Specify language, constraints |
| Comparison | Compare {a} and {b}:\nSimilarities: | Define comparison criteria |
| Explanation | Explain {concept} simply:\nExplanation: | Specify audience level |

*Figure: Prompt Templates for Common Research Tasks*

```
# Prompt template library for open models
class PromptTemplates:

    @staticmethod
    def summarize(text, max_words=100):
        return f'''Summarize the following text in approximately
{max_words} words.

Text: {text}

Summary:'''

    @staticmethod
    def classify(text, labels):
        label_str = ', '.join(labels)
```

```
        return f'''Classify the following text into one of these
categories: {label_str}.

Text: "{text}"

Category:'''

    @staticmethod
    def qa(context, question):
        return f'''Answer the question based only on the provided context.
If the answer is not in the context, say "Not found in context."

Context: {context}

Question: {question}

Answer:'''

    @staticmethod
    def few_shot(examples, query, task_description=''):
        prompt = task_description + '\n\n' if task_description else ''
        for inp, out in examples:
            prompt += f'Input: {inp}\nOutput: {out}\n\n'
        prompt += f'Input: {query}\nOutput:'
        return prompt

# Usage
prompt = PromptTemplates.classify(
    'The new GPU achieves 3x faster training speeds.',
    ['Technology', 'Science', 'Business', 'Sports']
)
print(prompt)
```

## Prompt Optimization Techniques

| Technique | Description | When to Use | Improvement |
|---|---|---|---|
| Role prompting | Assign a persona to the model | Complex reasoning tasks | 10-30% |
| Chain-of-thought | Ask model to explain reasoning | Math, logic, analysis | 20-50% |

| Technique | Description | When to Use | Improvement |
|---|---|---|---|
| Step-by-step | Break task into numbered steps | Multi-step processes | 15-40% |
| Output formatting | Specify JSON/XML/table format | Structured outputs | Reduces parsing errors |
| Negative examples | Show what NOT to do | Common failure modes | 10-20% |
| Self-consistency | Generate multiple, take majority | Ambiguous tasks | 5-15% |
| Prompt chaining | Break into sequential prompts | Complex workflows | 20-40% |

*Figure: Prompt Optimization Techniques and Expected Improvements*

# Chapter 18. Using the Hugging Face Hub Like a Researcher

## 18.1 Finding the Right Model

| Search Strategy | How to Use | Best For |
|---|---|---|
| Task filter | Models > Filter by task | Finding models for specific tasks |
| Download count | Sort by Most Downloads | Finding popular, tested models |
| Leaderboard | Open LLM Leaderboard | Comparing LLM performance |
| Library filter | Filter by PyTorch/TF/JAX | Framework compatibility |
| License filter | Filter by license type | Commercial vs research use |
| Search keywords | Search by name or keyword | Finding specific architectures |

*Figure: Model Search Strategies on the Hub*

## 18.2 Reading Model Cards Effectively

## 18.3 Uploading Your Models

```
from huggingface_hub import HfApi
```

```
api = HfApi()

# Push model to Hub
model.push_to_hub('your-username/my-fine-tuned-model')
tokenizer.push_to_hub('your-username/my-fine-tuned-model')

# Or upload specific files
api.upload_file(
    path_or_fileobj='results.json',
    path_in_repo='eval_results.json',
    repo_id='your-username/my-fine-tuned-model',
)
```

## 18.4 Version Comparison

### 🔬 LAB: Publish a Model to the Hub

Fine-tune a small model, create a comprehensive model card, and push it to the Hugging Face Hub.

# Chapter 19. Sharing, Demoing, and Publishing Results

## 19.1 Exporting and Cleaning Notebooks

## 19.2 Sharing on the Hub

## 19.3 Building Demos with Gradio

```
import gradio as gr
from transformers import pipeline

classifier = pipeline('sentiment-analysis', device=0)

def analyze_sentiment(text):
    result = classifier(text)[0]
    return {result['label']: result['score'],
            'POSITIVE' if result['label'] == 'NEGATIVE' else 'NEGATIVE': 1
- result['score']}

demo = gr.Interface(
    fn=analyze_sentiment,
    inputs=gr.Textbox(label='Enter text', lines=3),
    outputs=gr.Label(label='Sentiment'),
    title='Sentiment Analyzer',
    examples=['I love this product!', 'Terrible experience.'],
)
demo.launch(share=True)  # Creates public URL
```

## 19.4 Deploying to Hugging Face Spaces

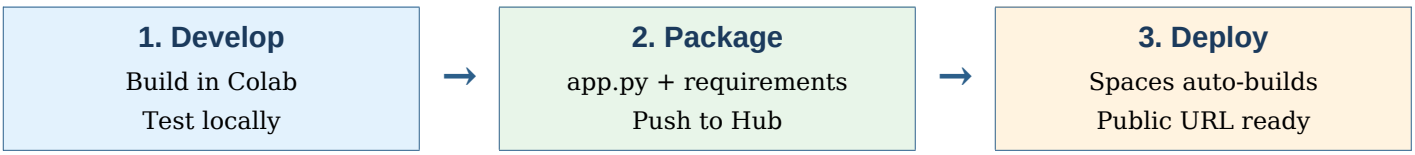


*Figure: Colab to Spaces deployment workflow*

## 🔬 LAB: Create and Deploy a Gradio Demo

Build an interactive demo for one of your fine-tuned models and deploy it to Hugging Face Spaces.

## Creating Professional Research Demos

A well-designed demo can make your research accessible to a much broader audience. The key principles of effective demos are simplicity, responsiveness, and clear documentation. Here is a more advanced Gradio example that incorporates best practices.

```
import gradio as gr
from transformers import pipeline
import time

# Load models once at startup
sentiment = pipeline('sentiment-analysis', device=0)
summarizer = pipeline('summarization', model='facebook/bart-large-cnn',
device=0)
ner = pipeline('ner', aggregation_strategy='simple', device=0)

def analyze_text(text, tasks):
    results = {}

    if 'Sentiment' in tasks:
        start = time.time()
        res = sentiment(text[:512])[0]
        results['Sentiment'] = f'{res["label"]} ({res["score"]:.4f})
[{time.time()-start:.2f}s]'

    if 'Summary' in tasks:
        start = time.time()
        if len(text.split()) > 50:
            res = summarizer(text[:1024], max_length=100)[0]
            results['Summary'] = f'{res["summary_text"]} [{time.time()-
start:.2f}s]'
        else:
            results['Summary'] = 'Text too short to summarize'

    if 'Named Entities' in tasks:
        start = time.time()
        entities = ner(text[:512])
        ent_str = ', '.join([f'{e["word"]} ({e["entity_group"]})' for e in
entities])
        results['Named Entities'] = f'{ent_str} [{time.time()-
start:.2f}s]'
```

```
    output = '\n\n'.join([f'**{k}:**\n{v}' for k, v in results.items()])
    return output

demo = gr.Interface(
    fn=analyze_text,
    inputs=[
        gr.Textbox(label='Input Text', lines=8,
                   placeholder='Enter text to analyze...'),
        gr.CheckboxGroup(
            ['Sentiment', 'Summary', 'Named Entities'],
            label='Analysis Tasks',
            value=['Sentiment']),
    ],
    outputs=gr.Markdown(label='Results'),
    title='Multi-Task NLP Analyzer',
    description='Analyze text using state-of-the-art Hugging Face
models.',
    examples=[
        ['Artificial intelligence is transforming healthcare...',
['Sentiment', 'Summary']],
        ['Dr. Sarah Chen from MIT published a paper on quantum
computing.', ['Named Entities']],
    ],
    theme=gr.themes.Soft(),
)

demo.launch(share=True)
```

## Deploying to Hugging Face Spaces: Step by Step

| Step | Action | Details |
|---|---|---|
| 1 | Create Space on Hub | huggingface.co/new-space, choose Gradio SDK |
| 2 | Create app.py | Main application file with Gradio interface |
| 3 | Create requirements.txt | List all dependencies with versions |
| 4 | Push to Hub | git push or upload via web interface |
| 5 | Wait for build | Space auto-builds and deploys (~2-5 minutes) |

| Step | Action | Details |
|---|---|---|
| 6 | Test and iterate | Update files to redeploy automatically |
| 7 | Share URL | https://huggingface.co/spaces/username/app-name |

*Figure: Step-by-step guide to deploying on Hugging Face Spaces*

## Part V Capstone Lab

### 🔬 LAB: End-to-End RAG Research Paper Q&A System

Build a complete RAG system that can answer questions about a collection of AI research paper abstracts. This lab integrates embeddings, retrieval, generation, prompt engineering, and evaluation from Chapters 16 through 19.

**Duration:** 90–120 minutes

```
# Part V Capstone: Research Paper Q&A System
from sentence_transformers import SentenceTransformer
from transformers import pipeline
import faiss, numpy as np, time

# Knowledge base: AI research paper abstracts
papers = [
    {'title': 'Attention Is All You Need', 'abstract': 'We propose a new network architecture, the Transformer, based solely on attention mechanisms, dispensing with recurrence and convolutions entirely. The Transformer allows for significantly more parallelization and can reach a new state of the art in translation quality.'},
    {'title': 'BERT', 'abstract': 'We introduce BERT, a bidirectional transformer pre-trained on unlabeled text. BERT obtains new state-of-the-art results on eleven natural language processing tasks, including pushing the GLUE score to 80.5 percent.'},
    {'title': 'GPT-3', 'abstract': 'We demonstrate that scaling up language models greatly improves task-agnostic, few-shot performance, sometimes even reaching competitiveness with prior state-of-the-art fine-tuning approaches.'},
    {'title': 'ResNet', 'abstract': 'We present a residual learning framework to ease the training of networks that are substantially deeper
```

```
than those used previously. We explicitly reformulate the layers as
learning residual functions with reference to the layer inputs.'},
    {'title': 'LoRA', 'abstract': 'We propose Low-Rank Adaptation, or
LoRA, which freezes the pre-trained model weights and injects trainable
rank decomposition matrices into each layer of the Transformer
architecture, greatly reducing the number of trainable parameters for
downstream tasks.'},
    {'title': 'Stable Diffusion', 'abstract': 'We apply diffusion models
in the latent space of powerful pretrained autoencoders, achieving a near-
optimal balance between complexity reduction and detail preservation,
yielding high-fidelity image synthesis.'},
    {'title': 'CLIP', 'abstract': 'We demonstrate that the simple pre-
training task of predicting which caption goes with which image is an
efficient and scalable way to learn SOTA image representations from
scratch on a dataset of 400 million image-text pairs.'},
    {'title': 'Whisper', 'abstract': 'We study the capabilities of speech
processing systems trained simply to predict large amounts of transcripts
of audio from the internet. When scaled to 680,000 hours of multilingual
supervision, the resulting models generalize well to standard
benchmarks.'},
    {'title': 'DPO', 'abstract': 'We introduce Direct Preference
Optimization, a simple algorithm that implicitly optimizes the same
objective as existing RLHF algorithms but is simple to implement and
straightforward to train, eliminating the need for a reward model.'},
    {'title': 'Mamba', 'abstract': 'We introduce Mamba, a selective state
space model architecture that achieves strong performance on information-
dense data such as language modeling, while scaling linearly in sequence
length.'},
]

# Build index
embedder = SentenceTransformer('all-MiniLM-L6-v2')
abstracts = [p['abstract'] for p in papers]
doc_embs = embedder.encode(abstracts, convert_to_numpy=True)
faiss.normalize_L2(doc_embs)
index = faiss.IndexFlatIP(doc_embs.shape[1])
index.add(doc_embs)
print(f'Indexed {len(papers)} papers')

# Generator
generator = pipeline('text2text-generation', model='google/flan-t5-base',
device=0)
```

```
def ask(question, k=3):
    # Retrieve
    q_emb = embedder.encode([question], convert_to_numpy=True)
    faiss.normalize_L2(q_emb)
    scores, ids = index.search(q_emb, k)

    # Build context
    context_parts = []
    print(f'\nRetrieved papers:')
    for idx, score in zip(ids[0], scores[0]):
        paper = papers[idx]
        context_parts.append(f'{paper["title"]}: {paper["abstract"]}')
        print(f'  [{score:.3f}] {paper["title"]}')
    context = '\n'.join(context_parts)

    # Generate
    prompt = f'''Based on the following research paper abstracts, answer
the question.
If the answer is not in the provided context, say so.

Context:
{context}

Question: {question}
Answer:'''
    answer = generator(prompt, max_length=200)[0]['generated_text']
    return answer

# Test queries
questions = [
    'What architecture does the Transformer use instead of recurrence?',
    'How does LoRA reduce the number of trainable parameters?',
    'What is the key innovation in Mamba compared to Transformers?',
    'How many hours of audio was Whisper trained on?',
    'What does DPO eliminate the need for?',
]

print('\n' + '=' * 60)
print('RESEARCH PAPER Q&A SYSTEM')
print('=' * 60)
for q in questions:
```

```
    print(f'\nQ: {q}')
    answer = ask(q)
    print(f'A: {answer}')
```

| Question | Retrieved Papers | Answer Quality (1-5) | Factually Correct? |
|---|---|---|---|
| Transformer architecture question | (Record top paper) | (Rate) | Yes / No |
| LoRA parameter reduction | (Record top paper) | (Rate) | Yes / No |
| Mamba vs Transformers | (Record top paper) | (Rate) | Yes / No |
| Whisper training data size | (Record top paper) | (Rate) | Yes / No |
| DPO innovation | (Record top paper) | (Rate) | Yes / No |

*Figure: Part V Capstone RAG Evaluation Results*

# PART VI

## PRACTICAL GUIDANCE

# Chapter 20. Common Errors and Debugging in Google Colab

## 20.1 Error Reference Table

| Error | Cause | Quick Fix |
|---|---|---|
| CUDA out of memory | Model + data exceeds GPU RAM | Reduce batch size, enable fp16, quantize |
| RuntimeError: CUDA error | GPU crash or driver issue | Restart runtime, reduce memory |
| ModuleNotFoundError | Package not installed | !pip install package_name |
| TokenizationError | Wrong tokenizer for model | Use AutoTokenizer.from_pretrained() |
| ValueError: too many values | Dataset format mismatch | Check column names, remove unused |
| ConnectionError (Hub) | Auth or network issue | login(), check token permissions |
| Session disconnected | Idle timeout or limit | Save to Drive, use checkpoints |
| Kernel died | RAM exceeded | Reduce data size, use streaming |
| NCCL error | Multi-GPU config issue | Use device_map='auto' |
| Slow training | CPU instead of GPU | Check runtime type, model.to('cuda') |

*Figure: Complete Error Reference Guide*

## 20.2 Debugging Strategies

```
# Memory debugging toolkit
import torch

def gpu_memory_report():
    if torch.cuda.is_available():
        allocated = torch.cuda.memory_allocated() / 1e9
        reserved = torch.cuda.memory_reserved() / 1e9
        total = torch.cuda.get_device_properties(0).total_mem / 1e9
        print(f'GPU Memory: {allocated:.2f} GB allocated / '
              f'{reserved:.2f} GB reserved / {total:.1f} GB total')
```

```
gpu_memory_report()

# Shape debugging
def debug_shapes(**tensors):
    for name, t in tensors.items():
        if hasattr(t, 'shape'):
            print(f'{name:>20}: shape={t.shape}, dtype={t.dtype},
device={t.device}')
```

## 20.3 Performance Optimization Checklist

| Check | Command | Expected Result |
|---|---|---|
| GPU enabled? | !nvidia-smi | Shows GPU info |
| Model on GPU? | next(model.parameters()).device | cuda:0 |
| FP16 enabled? | training_args.fp16 | True |
| Batch size optimal? | Monitor GPU memory | 80-90% utilization |
| No grad in inference? | torch.no_grad() context | Less memory used |
| Cache cleared? | torch.cuda.empty_cache() | After each experiment |

*Figure: Performance Optimization Checklist*

### 🔬 LAB: Debugging Challenge

Given a notebook with intentional errors (wrong tokenizer, missing GPU transfer, memory leak), identify and fix all issues.

## The Debugging Mindset

Effective debugging is a skill that separates productive researchers from frustrated ones. The key principle is to approach errors systematically rather than randomly trying fixes. Every error has a cause, and the error message almost always contains the information you need to find that cause.

When you encounter an error, follow this systematic process: First, read the complete error traceback from bottom to top. The last line tells you the type of error, and the lines above trace back to the source in your code. Second, identify the specific line in your code that triggered the error. Third, form a hypothesis about what went wrong. Fourth, test your hypothesis with a minimal example. Fifth, apply the fix and verify it works.

## CUDA Out of Memory: A Deep Dive

CUDA out-of-memory (OOM) errors are the most common issue when working with deep learning models in Colab. Understanding why they happen and having a systematic approach to resolving them will save you hours of frustration throughout your research career.

| Memory Consumer | Typical Size (7B model) | Can Be Reduced? | How |
|---|---|---|---|
| Model weights | 14 GB (FP16) | Yes | Quantization (4-bit: 3.5 GB) |
| Optimizer states | 14 GB (Adam, FP16) | Partially | 8-bit Adam, PEFT |
| Gradients | 14 GB (FP16) | Yes | Gradient checkpointing, PEFT |
| Activations | 2-10 GB (varies) | Yes | Gradient checkpointing, smaller batch |
| Input batch | 0.1-2 GB | Yes | Smaller batch size |
| CUDA context | ~0.5 GB | No | Fixed overhead |

*Figure: GPU Memory Breakdown for a 7B Parameter Model*

```
# Systematic OOM debugging script
import torch

def diagnose_oom():
    """Run this after an OOM error to understand what happened."""
    if not torch.cuda.is_available():
        print('No GPU available!')
```

```
        return

    total = torch.cuda.get_device_properties(0).total_mem / 1e9
    alloc = torch.cuda.memory_allocated() / 1e9
    cached = torch.cuda.memory_reserved() / 1e9

    print(f'GPU Memory Diagnosis:')
    print(f'  Total:     {total:.1f} GB')
    print(f'  Allocated: {alloc:.2f} GB ({alloc/total*100:.0f}%)')
    print(f'  Cached:    {cached:.2f} GB ({cached/total*100:.0f}%)')
    print(f'  Free:      {total-cached:.2f} GB')
    print()
    print('Recommended actions:')
    if alloc > total * 0.9:
        print('  [CRITICAL] > 90% memory used!')
        print('  - Reduce batch_size by half')
        print('  - Enable fp16=True')
        print('  - Use gradient_checkpointing=True')
        print('  - Try 4-bit quantization')
    elif alloc > total * 0.7:
        print('  [WARNING] > 70% memory used')
        print('  - Consider reducing batch_size')
        print('  - Enable gradient_accumulation_steps=2')
    else:
        print('  [OK] Memory usage is reasonable')

diagnose_oom()
```

## Common Dependency Issues and Solutions

The rapidly evolving Python machine learning ecosystem means that library incompatibilities are frequent. Here is a reference table of known compatibility issues and their solutions:

| Library Combination | Issue | Solution |
|---|---|---|
| transformers + torch | CUDA version mismatch | Use Colab's pre-installed torch |
| bitsandbytes on CPU | No CUDA support error | Only works with GPU; check runtime type |
| peft + older | API changes | pip install peft transformers --upgrade |

| Library Combination | Issue | Solution |
|---|---|---|
| transformers | | |
| sentence-transformers + torch 2.x | Deprecated functions | pip install sentence-transformers>=2.3 |
| accelerate + multi-GPU | NCCL errors in Colab | Use device_map='auto' instead |
| gradio + Colab | Port conflicts | Use share=True for public URL |
| tokenizers + fast tokenizer | Slow tokenizer warning | Set use_fast=True (default) |

*Figure: Known Library Compatibility Issues and Solutions*

# Chapter 21. Best Practices for AI Researchers

## 21.1 Research Workflow Habits

| Phase | Best Practice | Tool/Technique |
|---|---|---|
| Planning | Define hypothesis before experimenting | Research journal |
| Setup | Pin versions, set seeds, document config | requirements.txt, set_seed() |
| Data | Never touch test set during development | train_test_split() |
| Training | One variable at a time | Experiment tracker |
| Evaluation | Report mean +/- std over 3+ runs | evaluate library |
| Sharing | Include model card and code | Hugging Face Hub |
| Archival | Save everything to Drive | Google Drive mount |

*Figure: Research Workflow Best Practices*

## 21.2 Documentation Standards

## 21.3 Experiment Discipline

## 21.4 Efficiency Tips for Colab

| Tip | Time Saved | Effort Required |
|---|---|---|
| Use smallest model that validates your idea | Hours of GPU time | Low |
| Debug with 100-example subset first | Minutes per iteration | Low |
| Precompute embeddings and cache to Drive | Avoids re-computation | Medium |
| Use pipeline for quick experiments | Minutes of setup | Low |
| Learn Colab keyboard shortcuts | Seconds per action | Low |
| Create reusable utility functions | Minutes per notebook | Medium |
| Monitor GPU memory proactively | Avoids OOM crashes | Low |

*Figure: Efficiency Tips Ranked by Impact*

# Chapter 22. From Notebook to Real Project

## 22.1 Notebook to Script Conversion

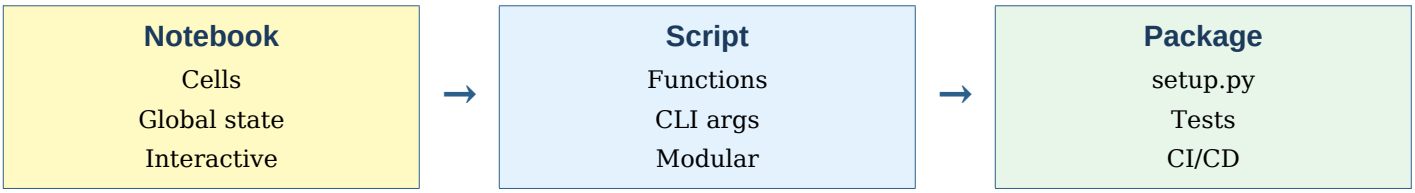


*Figure: Evolution from notebook to production code*

```
# Example: Converting notebook code to a reusable script
# train.py
import argparse
from transformers import (AutoModelForSequenceClassification,
    AutoTokenizer, Trainer, TrainingArguments)
from datasets import load_dataset

def parse_args():
    parser = argparse.ArgumentParser()
    parser.add_argument('--model_name', default='distilbert-base-uncased')
    parser.add_argument('--dataset', default='imdb')
    parser.add_argument('--lr', type=float, default=2e-5)
    parser.add_argument('--epochs', type=int, default=3)
    parser.add_argument('--batch_size', type=int, default=16)
    parser.add_argument('--output_dir', default='./results')
    return parser.parse_args()

def main():
    args = parse_args()
    tokenizer = AutoTokenizer.from_pretrained(args.model_name)
    model = AutoModelForSequenceClassification.from_pretrained(
        args.model_name, num_labels=2)
    dataset = load_dataset(args.dataset)
    # ... rest of training code ...

if __name__ == '__main__':
    main()
```

## 22.2 Project Structure

| Directory/File | Purpose |
|---|---|
| src/ | Source code modules |
| src/data.py | Data loading and preprocessing |
| src/model.py | Model definition and configuration |
| src/train.py | Training script with CLI |
| src/evaluate.py | Evaluation and metrics |
| configs/ | YAML/JSON configuration files |
| tests/ | Unit and integration tests |
| notebooks/ | Exploration notebooks |
| requirements.txt | Pinned dependencies |
| README.md | Project documentation |
| Makefile | Common commands (train, eval, test) |

*Figure: Recommended Project Structure*

## 22.3 When Colab Is No Longer Enough

| Signal | Alternative | Cost Range |
|---|---|---|
| Training exceeds 12h session limit | Cloud VM (GCP, AWS) | $1-5/hour |
| Model doesn't fit in 15GB VRAM | A100 cloud instance | $2-5/hour |
| Need multi-GPU training | Multi-GPU cloud setup | $5-30/hour |
| Need persistent background jobs | Dedicated server / Lambda | $0.50-2/hour |
| Reproducible CI/CD pipeline | GitHub Actions + cloud | Variable |
| Production serving | Inference endpoints / K8s | Variable |

*Figure: When to Graduate from Colab*

### 🔬 LAB: Convert a Notebook to a Production Script

Take the fine-tuning notebook from Chapter 12 and convert it into a modular, CLI-driven Python script with configuration files and unit tests.

## Testing Your ML Code

Testing machine learning code is different from testing traditional software because the outputs are probabilistic rather than deterministic. However, there are still many aspects of your code that can and should be tested systematically.

| Test Type | What to Test | Example |
|---|---|---|
| Unit test | Individual functions | Tokenization output shape |
| Data test | Data loading and preprocessing | Check for NaN, correct types |
| Model test | Model loads and runs | Forward pass produces output |
| Shape test | Tensor dimensions match | Output shape matches num_labels |
| Regression test | Metrics don't decrease | Accuracy >= baseline threshold |
| Integration test | End-to-end pipeline | Train 1 step without errors |
| Smoke test | Basic sanity checks | Model produces non-zero output |

*Figure: Types of Tests for Machine Learning Code*

```
# Example: Simple test suite for ML project
import unittest
import torch
from transformers import AutoTokenizer, AutoModelForSequenceClassification

class TestModel(unittest.TestCase):
    @classmethod
    def setUpClass(cls):
        cls.model_name = 'distilbert-base-uncased'
        cls.tokenizer = AutoTokenizer.from_pretrained(cls.model_name)
        cls.model = AutoModelForSequenceClassification.from_pretrained(
            cls.model_name, num_labels=2)

    def test_tokenizer_output_shape(self):
        inputs = self.tokenizer('Hello world', return_tensors='pt')
        self.assertEqual(inputs['input_ids'].dim(), 2)
        self.assertEqual(inputs['input_ids'].shape[0], 1)

    def test_model_forward_pass(self):
        inputs = self.tokenizer('Test input', return_tensors='pt')
```

```
        with torch.no_grad():
            outputs = self.model(**inputs)
        self.assertEqual(outputs.logits.shape, (1, 2))

    def test_model_produces_probabilities(self):
        inputs = self.tokenizer('Test input', return_tensors='pt')
        with torch.no_grad():
            outputs = self.model(**inputs)
        probs = torch.softmax(outputs.logits, dim=-1)
        self.assertAlmostEqual(probs.sum().item(), 1.0, places=5)

    def test_batch_processing(self):
        texts = ['First text', 'Second text', 'Third text']
        inputs = self.tokenizer(texts, return_tensors='pt',
                                padding=True, truncation=True)
        with torch.no_grad():
            outputs = self.model(**inputs)
        self.assertEqual(outputs.logits.shape[0], 3)

# Run tests
# In Colab: !python -m pytest test_model.py -v
# Or: unittest.main(argv=[''], exit=False)
```

## Scaling Beyond Colab: A Practical Guide

When your research outgrows Colab, you need a plan for transitioning to more powerful infrastructure. The good news is that the skills and code you have developed throughout this book transfer directly to any Python environment with GPU support. The main differences are in environment management, job scheduling, and cost optimization.

| Platform | Setup Difficulty | Cost/Hour (1xA100) | Best For | Min Commitment |
|---|---|---|---|---|
| Google Colab Pro+ | None | ~$0.12 | Prototyping, education | $50/month |
| Lambda Cloud | Low | ~$1.10 | Simple GPU jobs | Per-second billing |
| Vast.ai | Low | ~$0.80 | Budget GPU rental | Per-second billing |
| RunPod | Low | ~$0.74 | Serverless GPU | Per-second billing |

| Platform | Setup Difficulty | Cost/Hour (1xA100) | Best For | Min Commitment |
|---|---|---|---|---|
| AWS SageMaker | Medium | ~$4.00 | Enterprise ML ops | Per-second billing |
| Google Cloud (GCP) | Medium | ~$3.67 | GCP ecosystem users | Per-second billing |
| Azure ML | Medium | ~$3.40 | Azure ecosystem users | Per-second billing |
| Own Hardware | High | $0 (after purchase) | Long-term heavy use | $8,000–20,000 upfront |

*Figure: GPU Computing Platform Comparison for Scaling Beyond Colab*

# References and Further Reading

The following references provide the foundational knowledge and advanced topics covered in this handbook. They are organized by category for easy reference.

## Foundational Papers

**[1]** Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., & Polosukhin, I. (2017). Attention Is All You Need. Advances in Neural Information Processing Systems, 30 (NeurIPS 2017).

**[2]** Devlin, J., Chang, M.-W., Lee, K., & Toutanova, K. (2019). BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. Proceedings of NAACL-HLT 2019.

**[3]** Radford, A., Wu, J., Child, R., Luan, D., Amodei, D., & Sutskever, I. (2019). Language Models are Unsupervised Multitask Learners. OpenAI Technical Report.

**[4]** Brown, T., Mann, B., Ryder, N., et al. (2020). Language Models are Few-Shot Learners. Advances in Neural Information Processing Systems, 33 (NeurIPS 2020).

**[5]** Raffel, C., Shazeer, N., Roberts, A., Lee, K., Narang, S., Matena, M., Zhou, Y., Li, W., & Liu, P. J. (2020). Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer. Journal of Machine Learning Research, 21(140), 1–67.

**[6]** Lewis, M., Liu, Y., Goyal, N., et al. (2020). BART: Denoising Sequence-to-Sequence Pre-training for Natural Language Generation, Translation, and Comprehension. Proceedings of ACL 2020.

## Efficient Training and Fine-Tuning

**[7]** Hu, E. J., Shen, Y., Wallis, P., Allen-Zhu, Z., Li, Y., Wang, S., Wang, L., & Chen, W. (2022). LoRA: Low-Rank Adaptation of Large Language Models. Proceedings of ICLR 2022.

**[8]** Dettmers, T., Pagnoni, A., Holtzman, A., & Zettlemoyer, L. (2023). QLoRA: Efficient Finetuning of Quantized Language Models. Advances in Neural Information Processing Systems, 36 (NeurIPS 2023).

**[9]** Dettmers, T., Lewis, M., Belkada, Y., & Zettlemoyer, L. (2022). LLM.int8(): 8-bit Matrix Multiplication for Transformers at Scale. Advances in Neural Information Processing Systems, 35.

**[10]** Dao, T., Fu, D. Y., Ermon, S., Rudra, A., & Ré, C. (2022). FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness. Advances in Neural Information Processing Systems, 35.

## Computer Vision

**[11]** Dosovitskiy, A., Beyer, L., Kolesnikov, A., et al. (2021). An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale. Proceedings of ICLR 2021.

**[12]** Radford, A., Kim, J. W., Hallacy, C., et al. (2021). Learning Transferable Visual Models From Natural Language Supervision. Proceedings of ICML 2021.

**[13]** Rombach, R., Blattmann, A., Lorenz, D., Esser, P., & Ommer, B. (2022). High-Resolution Image Synthesis with Latent Diffusion Models. Proceedings of CVPR 2022.

**[14]** Carion, N., Massa, F., Synnaeve, G., Usunier, N., Kirillov, A., & Zagoruyko, S. (2020). End-to-End Object Detection with Transformers. Proceedings of ECCV 2020.

## Speech and Audio

**[15]** Radford, A., Kim, J. W., Xu, T., Brockman, G., McLeavey, C., & Sutskever, I. (2023). Robust Speech Recognition via Large-Scale Weak Supervision. Proceedings of ICML 2023.

**[16]** Baevski, A., Zhou, Y., Mohamed, A., & Auli, M. (2020). wav2vec 2.0: A Framework for Self-Supervised Learning of Speech Representations. Advances in Neural Information Processing Systems, 33.

## Retrieval-Augmented Generation

**[17]** Lewis, P., Perez, E., Piktus, A., et al. (2020). Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks. Advances in Neural Information Processing Systems, 33.

**[18]** Reimers, N., & Gurevych, I. (2019). Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks. Proceedings of EMNLP 2019.

**[19]** Johnson, J., Douze, M., & Jégou, H. (2019). Billion-scale similarity search with GPUs. IEEE Transactions on Big Data, 7(3), 535–547.

## Large Language Models

**[20]** Touvron, H., Lavril, T., Izacard, G., et al. (2023). LLaMA: Open and Efficient Foundation Language Models. arXiv preprint arXiv:2302.13971.

**[21]** Jiang, A. Q., Sablayrolles, A., Mensch, A., et al. (2023). Mistral 7B. arXiv preprint arXiv:2310.06825.

**[22]** Touvron, H., Martin, L., Stone, K., et al. (2023). LLaMA 2: Open Foundation and Fine-Tuned Chat Models. arXiv preprint arXiv:2307.09288.

## Evaluation and Benchmarks

**[23]** Wang, A., Singh, A., Michael, J., Hill, F., Levy, O., & Bowman, S. (2019). GLUE: A Multi-Task Benchmark and Analysis Platform for Natural Language Understanding. Proceedings of ICLR 2019.

**[24]** Rajpurkar, P., Zhang, J., Lopyrev, K., & Liang, P. (2016). SQuAD: 100,000+ Questions for Machine Comprehension of Text. Proceedings of EMNLP 2016.

**[25]** Papineni, K., Roukos, S., Ward, T., & Zhu, W.-J. (2002). BLEU: A Method for Automatic Evaluation of Machine Translation. Proceedings of ACL 2002.

## Hugging Face Documentation and Resources

**[26]** Wolf, T., Debut, L., Sanh, V., et al. (2020). Transformers: State-of-the-Art Natural Language Processing. Proceedings of EMNLP 2020: System Demonstrations, 38–45.

**[27]** Lhoest, Q., Villanova del Moral, A., Jernite, Y., et al. (2021). Datasets: A Community Library for Natural Language Processing. Proceedings of EMNLP 2021: System Demonstrations.

**[28]** Hugging Face. (2024). Transformers Documentation. https://huggingface.co/docs/transformers

**[29]** Hugging Face. (2024). Datasets Documentation. https://huggingface.co/docs/datasets

**[30]** Hugging Face. (2024). PEFT Documentation. https://huggingface.co/docs/peft

**[31]** Hugging Face. (2024). The Hugging Face Course. https://huggingface.co/learn/nlp-course

## Books and Textbooks

**[32]** Tunstall, L., von Werra, L., & Wolf, T. (2022). Natural Language Processing with Transformers: Building Language Applications with Hugging Face. O'Reilly Media.

**[33]** Goodfellow, I., Bengio, Y., & Courville, A. (2016). Deep Learning. MIT Press. https://www.deeplearningbook.org

**[34]** Jurafsky, D., & Martin, J. H. (2024). Speech and Language Processing (3rd edition draft). https://web.stanford.edu/~jurafsky/slp3/

**[35]** Zhang, A., Lipton, Z. C., Li, M., & Smola, A. J. (2023). Dive into Deep Learning. Cambridge University Press. https://d2l.ai

## Online Resources

**[36]** Hugging Face Hub: https://huggingface.co — Central repository for models, datasets, and Spaces.

**[37]** Google Colab: https://colab.research.google.com — Free cloud notebook environment with GPU access.

**[38]** Papers With Code: https://paperswithcode.com — Machine learning papers with code and benchmarks.

**[39]** arXiv: https://arxiv.org — Open-access archive for scientific papers in AI and machine learning.

**[40]** Open LLM Leaderboard: https://huggingface.co/spaces/HuggingFaceH4/open_llm_leaderboard — Standardized LLM benchmarks.

# PART

# APPENDICES

# Appendix A. Essential Colab Shortcuts

| Shortcut | Action | Category |
|---|---|---|
| Ctrl + Enter | Run current cell | Execution |
| Shift + Enter | Run cell, move to next | Execution |
| Alt + Enter | Run cell, insert below | Execution |
| Ctrl + M, B | Insert code cell below | Navigation |
| Ctrl + M, A | Insert code cell above | Navigation |
| Ctrl + M, D | Delete current cell | Editing |
| Ctrl + M, M | Convert to Markdown cell | Editing |
| Ctrl + M, Y | Convert to code cell | Editing |
| Ctrl + / | Toggle comment | Editing |
| Ctrl + Shift + H | Open command palette | Navigation |
| Ctrl + S | Save notebook | File |
| Ctrl + Z | Undo | Editing |
| Ctrl + Shift + Z | Redo | Editing |
| Tab | Code completion | Coding |
| Shift + Tab | Show function signature | Coding |
| Ctrl + F | Find in notebook | Navigation |
| Ctrl + H | Find and replace | Navigation |

*Figure: Essential Google Colab Keyboard Shortcuts*

# Appendix B. Hugging Face Libraries Quick Reference

| Library | Install Command | Primary Use | Key Classes/Functions |
|---|---|---|---|
| transformers | pip install transformers | Model loading & inference | AutoModel, pipeline, Trainer |
| datasets | pip install datasets | Data loading & processing | load_dataset, Dataset.map |
| tokenizers | pip install tokenizers | Fast tokenization | Tokenizer, BPE, WordPiece |
| evaluate | pip install evaluate | Metric computation | evaluate.load, combine |
| accelerate | pip install accelerate | Distributed training | Accelerator, dispatch_model |
| peft | pip install peft | Efficient fine-tuning | LoraConfig, get_peft_model |
| bitsandbytes | pip install bitsandbytes | Quantization | BitsAndBytesConfig |
| sentence-transformers | pip install sentence-transformers | Embeddings | SentenceTransformer |
| diffusers | pip install diffusers | Image generation | StableDiffusionPipeline |
| gradio | pip install gradio | Interactive demos | gr.Interface, gr.Blocks |
| huggingface_hub | pip install huggingface_hub | Hub interaction | login, HfApi, push_to_hub |

*Figure: Hugging Face Library Quick Reference*

# Appendix C. Model Selection Checklist

| # | Question | Your Answer |
|---|---|---|
| 1 | What specific task do you need? (classification, generation, etc.) | |
| 2 | What is the minimum acceptable performance? | |
| 3 | How much GPU memory is available? | |
| 4 | What are your latency requirements? | |
| 5 | What language(s) must the model support? | |
| 6 | What license is acceptable? (MIT, Apache, research-only) | |
| 7 | Is the training data similar to your domain? | |
| 8 | Do you need to fine-tune or just run inference? | |
| 9 | Is the model actively maintained? | |
| 10 | Are there benchmark results you can compare against? | |

*Figure: Model Selection Checklist — Fill in for each project*

# Appendix D. Troubleshooting Cheatsheet

| Problem | Likely Cause | Solution |
|---|---|---|
| CUDA OOM | Model too large for GPU | Reduce batch, fp16, quantize, smaller model |
| Module not found | Package not installed | !pip install package; restart if needed |
| Tokenizer mismatch | Wrong tokenizer loaded | Use AutoTokenizer with same model name |
| NaN loss | Learning rate too high | Reduce to 1e-5 or 2e-5, check data |
| Slow training | Running on CPU | Runtime > Change type > GPU |
| Drive mount fails | Auth expired | Revoke and re-authorize Google account |
| Hub auth error | Invalid/expired token | Generate new token at hf.co/settings |
| Kernel crash | RAM exceeded | Use streaming, smaller dataset, gc.collect() |
| Import error after install | Runtime not restarted | Restart runtime, reinstall |
| Wrong predictions | Model in training mode | Call model.eval() before inference |
| Checkpoint not found | Session expired | Save to Drive, use resume_from_checkpoint |
| NCCL timeout | Multi-device issue | Set device_map='auto' or single GPU |

*Figure: Complete Troubleshooting Cheatsheet*

---

*End of Book*

Thank you for reading the Hugging Face Handbook.
Happy researching!